Efficient Scheduling Regulation-Abiding Geo-Distributed Data Analytics Jobs

Xiaoda Zhang, Zhuzhong Qian, Sheng Zhang, Yize Li, Xiangbo Li, Xiaoliang Wang, Sanglu Lu
State Key Laboratory for Novel Software Technology, Nanjing University

***Abstract*—Geo-distributed data analytics are increasingly common to derive useful information in large organisations, but this requirement faces unique challenges including regulatory constraints, WAN bandwidth limits, and high monetary costs. Naive extension of existing cluster-scale systems to the scale of geo-distributed data centers fails to meet regulatory constraints. Our goal is to develop an regulation-abiding data analytics system that can guarantee efficient geo-distributed job performance economically.**

To this end, we present HOUTU, that is composed of multiple autonomous systems, each operating in a sovereign data center. HOUTU maintains a job manager (JM) for a geo-distributed job in each data center, so that these replicated JMs could *individually* and *cooperatively* manage resources and assign tasks. We bulid HOUTU by Spark, YARN, and Zookeeper as underlying blocks. Our experiments with typical workloads on the prototype running across four Alibaba Cloud regions show that HOUTU achieves around 30% performance (makespan and average job response time) improvement compared to the existing centralized architecture, but with only a quarter of monetary costs. We also theoretically prove the proposed methods guarantee $O(1)$-competitive ratio in terms of makespan when jobs arrive in an online manner.

***Index Terms*—data analytics, distributed systems, resource management, scheduling**

I. INTRODUCTION

Nowadays, organizations are deploying their services in multiple data centers around the world to meet the latency-sensitive requirements [9], [28]. As a result, the raw data – including user interaction logging, infrastructure monitoring, and service traces – is generated at geographically distributed data centers. Analytics jobs on these geo-distributed data are emerging as a daily requirement [18]–[20], [30], [31], [36]–[39].

Because these analytics jobs usually support the real-time decisions and online predictions, minimizing response time and maximizing throughput are important goals. For most organizations who have the geo-distributed data analytics requirement, the most convenient way is resorting to public cloud instances.

Existing researches are motivated by wide-area network (WAN) bandwidth limits, so they optimize tasks and/or data placement across data centers to improve data locality [19], [20], [30], [36]–[38]. However, there are two additional concerns that restrict many institutes to widely adopt cloud services. One concern is that all previous works employ a centralized architecture where a monolithic master controls the resources of the worker machines from all data centers, as shown in Fig. 1(a). We argue that regulatory constraints prevent us to do so, due to that more and more regions are establishing laws to restrict IT resources from being controlled by other untrusted parties in the shared environment [5], [33], [42] (§II-A). An alternative is to deploy an *autonomous* data analytics system per data center (Fig. 1(b)), and extend the functionalities of original system to allow to coordinate for geo-distributed jobs. The other concern is to the high monetary cost of deploying and maintaining cloud services [26], [44], [45]. To reduce the cost, besides carefully managing traffic through WAN, one simple idea is to purchase Spot instances to deploy the system serves, because prices of Spot instances are often significantly lower than fixed prices for the same instances with a reliable Service Level Agreement.

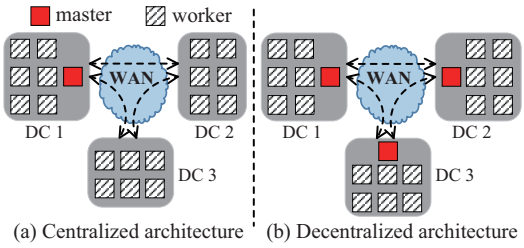


Fig. 1. Centralized vs. decentralized data analytics.

In this paper, we aim to *guarantee efficient geo-distributed job performance in the decentralized (regulation-abiding) architecture in an economic way.*

To achieve our goal, such a system needs to address three key challenges. First, we need to design an efficient task scheduling strategy in the decentralized architecture. This is challenging because data transmission rate across

System	Architecture	Application		Monetary reduction		Scheduling method	
	Centralized/Decentralized	General	Specific	WAN	Compute	LP-based	Others
Gaia [18]	-/ √	-	machine learning	√	-	-	local traffic aggreg. before global sync.
Tetrium [19]	√ /-	√		√	-	√	
SWAG [20]	√ /-	√		-	-	-	a variant of SRPT
Iridium [30]	√ /-	√		√	-	√	
Clarinet [36]	√ /-	-	SQL query	√	-	√	
Geode [37]	√ /-	-	SQL query	√	-	√	
Houtu	-/ √	√		√	√	-	utilization feedback + work stealing

Fig. 2. Overview of design features of existing geo-distributed data analytics systems compared to HOUTU.

data centers varies even in a short period (§II-B). Second, we need to find an online resource management mechanism for jobs. This is difficult because we neither assume job characteristics as a priori knowledge in the online scenario, unlike many alternatives [14]–[16], [25], nor use offline analysis [35] for its significant overhead. Third, we need to implement fault tolerance mechanism for jobs running atop unreliable Spot instances. Though existing frameworks [13], [22], [41] tolerate task-level failures, job-level fault tolerance is absent. While in the unreliable setting, the two types of failures have the same chance to occur.

In this work, we present HOUTU to achieve our goal. Fig. 2 surveys the design features. First, HOUTU supports general-type applications in its decentralized architecture. The key idea is to maintain a job manager (JM) for the geo-distributed job in each data center, and each JM can *individually* assign tasks within its own data center, and also can *cooperatively* assign tasks between data centers. HOUTU applies a new work stealing method when assigning tasks, which can adjust its assigning decisions according to the changeable environment (§IV). Next, HOUTU classifies three cases where each job manager *independently* either requests more resources, or maintains current resources, or proactively releases some resources for resource management. The key insight here is using nearly past resource utilization as feedback, irrespectively of the prediction of future job characteristics. Even without the future job characteristics, we theoretically prove the efficiency of job performance by extending the very recent result [43]. Finally, the worker machines in HOUTU are composed of Spot instances to reduce monetary cost of compute resources. Each replicated JM in a job keeps track of the current process of the job execution. We carefully design what need to be included in the job's intermediate information, which can be used to successfully recover the failure, of even the primary JM (§III).

We build HOUTU in Spark [41] on YARN [34] system, and leverage Zookeeper [21] to guarantee the intermediate information consistent among job managers in different data centers. We deploy HOUTU across four regions on Alibaba Cloud (AliCloud). Our evaluation with typical workloads shows that, HOUTU: (1) achieves efficient job performance as in the centralized architecture; (2) guarantees reliable job executions when facing job failures; and (3) is very effective in reducing monetary costs.

The rest of paper is organized as follows. We discuss the background and motivation in §II. The system overview of HOUTU is given in §III. Then, we describe the algorithms in §IV, the implementation in §V and evaluations in §VI. The related work is presented in §VII. Finally, we draw the conclusion in §VIII.

II. BACKGROUND AND MOTIVATION

A. Regulations prevent a master controlling everything

Public clouds allow users to instantiate virtual machines (instances) on demand. In turn, the use of virtualization allows third-party cloud providers to maximize the utilization of their sunk capital costs by multiplexing many customer instances across a shared physical infrastructure. However, this approach introduces new *vulnerabilities*. It is possible to mount cross-VM side-channel attacks to extract information from a target VM [12], [32]. Further attack amplifier can turn this initial compromise of a host into a platform for launching a broad, cloud-wide attack [11]. Hence, cloud providers and exiting works are proposing solutions in which a group of instances have their *external* connectivity restricted according to a declared policy as a defense against control leakage [2], [42].

	NC-3	NC-5	EC-1	SC-1
NC-3	(821,95)	(79,22)	(78,24)	(79,24)
NC-5	--	(820,115)	(103,28)	(71,28)
EC-1	--	--	(848,99)	(103,30)
SC-1	--	--	--	(821,107)

Fig. 3. Measured network bandwidth between four different regions in AliCloud. The entry is of form (Average, Standard deviation) Mbps.

	Reserved (per year)	On-demand (per hour)	Spot (per hour)
AliCloud [1]	866	0.312	0.036
EC2 [4]	1013	0.2	0.035
GCP [6]	1164	0.19	0.04
Azure [8]	1312	0.26	> 0.06

Fig. 4. Three pricing ways to pay for an instance with ⟨4 vCPU, 16 GB memory⟩ in GCP, EC2, AliCloud and Azure (in USD), respectively.

```
lineitem = textFile( "hdfs://master1:9000/tpch/lineitem.tbl" )
orders = textFile( "hdfs://master2:9000/tpch/orders.tbl" )
customer = textFile( "hdfs://master3:9000/tpch/customer.tbl" )
Shipping_Priority = sql( "SELECT L_ORDERKEY,
     SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS REVENUE,
     O_ORDERDATE, O_SHIPPRIORITY FROM customer C JOIN orders O
     ON C.C_CUSTKEY=O.O_CUSTKEY JOIN lineitem L ON
     L.L_ORDERKEY=O.O_ORDERKEY WHERE O_ORDERDATE < '1995-03-15'
     AND L_SHIPDATE > '1995-03-15' GROUP BY L_ORDERKEY,
     O_ORDERDATE, O_SHIPPRIORITY ORDER BY REVENUE DESC,
     O_ORDERDATE LIMIT 10" )
```

Fig. 5. Pseudo-code of a job's description in HOUTU.

By following exactly this guideline, we propose a decentralized architecture (Fig. 1(b)), in which worker machines in a data center are controlled by their own master machine, while masters from different data centers can interact, abiding regulatory constraints. We speculate that *derived* information, such as aggregates and reports (which are critical for business intelligence but have less dramatic privacy implications) may still be allowed to cross geographical boundaries.

B. Changeable environment requires adaptive methods

It is well known that WAN bandwidth is a very scarce and costly resource relative to LAN bandwidth. To quantify them, we measure the network bandwidth between all pairs of AliCloud in four regions including NorthChina-3 (NC-3), NorthChina-5 (NC-5), EastChina (EC-1), and SouthChina-1 (SC-1). We measure the network bandwidth of each pair of different regions for three rounds, each for 5 minutes. As shown in Fig. 3, the bandwidth within a data center is around 820 Mbps, while around 100 Mbps between data centers.

What we emphasize is that the WAN bandwidth *varies* between different regions even in a small period. The standard deviation can be as much as 30% of the available WAN bandwidth itself. The fluctuated bandwidth leads to data transmission time unpredictable [17], [24]. Furthermore, it may not always be the WAN bandwidth that causes runtime performance bottlenecks in wide-area data analytics. It is confirmed that memory may also becomes the bottleneck at runtime [38], thus these uncertainties lead LP-based task assigning methods [19], [30], [36], [37] sub-optimal. Instead, we design combinatorial task assigning method that can make online decisions to the changeable environment.

C. Employing Spot instance needs fault tolerance

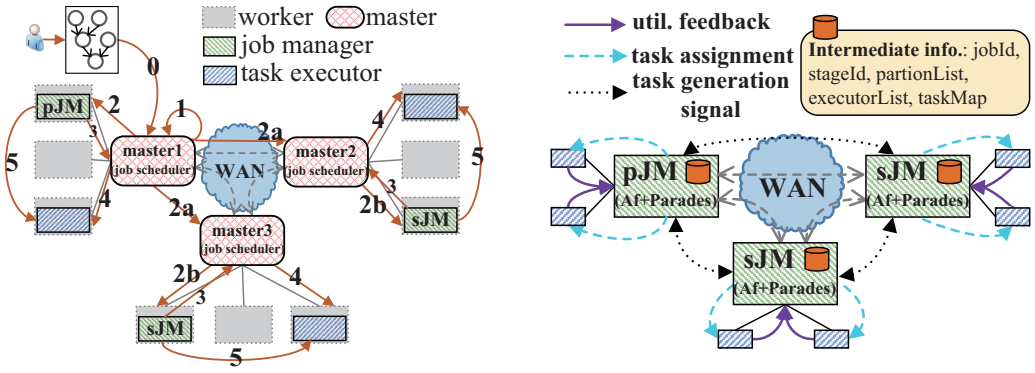
Besides offering reliable (Reserved and On-demand) instances, cloud providers such as Alibaba Cloud [1], Amazon EC2 [4], Google Cloud Platform (GCP) [6], and Microsoft Azure [8] also offer "Spot instances". Because the Spot instance market mechanism does not provide a way to guarantee how long an instance will run before it is terminated, Spot market prices are often significantly lower than fixed prices for the same instances with a reliability SLA (by up to 10x lower than On-demand price, and 3x lower than Reserved price, as an example shown in Fig. 4).

To guarantee reliable job executions using Spot instances with reduced cost in a systematic way, it needs to tolerate both job-level and task-level failures due to the terminations of unreliable instances, where the former refers to the failure of job managers. Because both job managers and tasks run in *unified containers*, the two types of failures have the same opportunity to occur when terminations of instances happen. Unfortunately, while current systems [13], [22], [41] tolerate the task-level failures, they do not tolerate job-level failures except for restarting the failed job. We implement the job-level fault-tolerance, which saves current process for each geo-distributed job across data centers, and a failed job manager will not affect the job's execution (§III-B).

III. System Overview

A. Houtu architecture and a job's lifecycle

As shown in Fig. 6(a), HOUTU is composed with several autonomous systems, deployed in geographically distributed data centers. Each system consists of a master node and multiple worker nodes. (Each node is a virtual machine instance in cloud.) As stated in §I, HOUTU is a general system that handles a wide range of analytics jobs without requiring any job description changes. The users can specify the data locations "as if" in a local



(a) HOUTU's architecture and a job's lifecycle.

(b) A job's *logical* topology in HOUTU.

Fig. 6. HOUTU's architecture, a job's lifecycle and a job's logical topology in it.

cluster, except with different "masters". In the SQL example shown in Fig. 5, three tables are dispersed over three data centers, and the job is to derive aggregate information from all these tables.

The units in resources scheduling is containers, which corresponding to some fixed amount of memory and cores, as in [15], [16], [30], [36]. Suppose a user submits a job to a chosen master (step 0). The job is abstracted as a directed acyclic graph (DAG), where each vertex represents a task and edges encode input-output dependencies. The master would first resolve the job's description to check whether it needs remote resources (step 1). The master directly generates a *primary* job manager (pJM) within its own cluster (step 2). For the remote resources, the master forwards the job's description to the remote masters (step 2a) and tells them to generate *semi-active* job manager (sJM) for the job (step 2b). Then, to obtain compute resources (*e.g.* task executors), the job managers independently send resource requests to their local masters (step 3). The masters (job schedulers) schedule resources to the JMs according to their own scheduling invariants, and return to the JMs containers (step 4). After that, the JMs send the tasks to run in the allocated resources (step 5). As the DAG job is dynamically unfolded and resource requests of the job usually are not satisfied in a single wave, JMs often repeat steps 3 – 5 for multiple times. JMs and tasks run in unified containers.

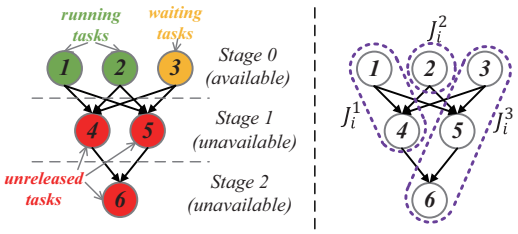
Throughout the execution course, regulatory constraints are respected since worker machines are controlled by the local master. We leave the design of how job managers request resources, and how to schedule tasks within and between data centers in §IV.

B. Normal operation and failure recovery of a job

1) Normal operation: When the master (to which a user submit the job) forwards the request (step 2a in Fig. 6(a)), it includes the job's description. Thus, each generated job manager holds the complete DAG structure of the job. When tasks in a job become available (*i.e.* all prerequisite tasks complete), the pJM and sJMs *cooperatively* schedule tasks to execute (dot line in Fig. 6(b)). We call each sJM semi-active because it has freedom to determine the task assignment in its own cluster (dash line), to coordinate with other sJMs about task assignment, and to manage its compute resources according to resource usage feedback (purple solid line).

After a task completes, it reports to its job manager (pJM or sJM) about the output partition location. The JM collects the partition location information in its cluster, modifies the *partitionList*, and then notifies other JMs to keep consistent of it. Besides the partitionList, HOUTU includes jobId, stageId, containerList (the available containers from all data centers), and taskMap (which task should be assigned by which JM) in a job's intermediate information (Fig. 6(b)). HOUTU maintains a replication of the intermediate information in each data center.

2) Failure recovery: As stated before, we focus on the recovery of job-level failures in this work. When a semi-active job manager fails because of the unpredictable termination of its host, the primary job manager will notice it and then send a request through its local master to generate a new sJM in the remote data center (like steps 2a and 2b in Fig. 6(a)). This sJM starts with the original job's description and the intermediate information in its cluster, and recognises its role (as semi-active). It *inherits* the containers belonging to the previous sJM, and *continues* to operate as in normal. If



(a) Example of a running DAG job (b) Sub-job partition of the job

Fig. 7. A running DAG job, and its partition across data centers.

the primary fails, the sJMs will elect a new primary using the consistent protocol (in Zookeeper). The new pJM updates and propagates the intermediate information about its role change. Next, the new primary *continues* the process of the job, operates in normal and generates a new semi-active JM to replace the failed pJM as above.

IV. Design

In this section, we first provide the problem statement and the key requirements for algorithm design. Next, we show how JMs use resource utilization feedback to manage resources (step 3 in Fig. 6(a)). Then, we describe how the JMs schedule tasks within and between data centers (step 5 in Fig. 6(a)). Finally, we theoretically analyze the performance of the algorithms.

A. Problem statement

Instead of assuming the priori knowledge of *complete* characteristics of jobs [14]–[16], [25] which restricts the workloads to recurring jobs, or exploiting offline analysis to train black-box models which incurs training overheads for each new workload [35], we aim to guarantee efficiency of general-type jobs in online setting. We rely on only *partial* priori knowledge of a job (the knowledge of available tasks). In the example of Fig. 7(a), only the task information (including the input data locations, fine-grained resource requirements, and process times) in *Stage 0* is currently known, while the task information in *Stage 1* and *Stage 2* is currently unknown.

In the scenario where multiple DAG jobs arrive and leave online, we are interested in minimizing the makespan and average job response time (*i.e.* the duration time from its release to its completion). Please refer to Appendix A for the complete problem formulation.

B. Design requirements

Requirement 1: The algorithms should balance WAN monetary cost and job performance. Purely minimizing job response time could result in increased WAN usage, leading to high monetary cost. Even worse, purely optimizing WAN usage can arbitrarily increase job response time. This is because WAN bandwidth cost savings are obtained by reducing WAN usage on all links, whereas job speedups are obtained (partially) by reducing WAN usage on the bottleneck link. Thus, to ensure fast job responses and reasonable bandwidth costs, the algorithms should incorporate a "knob" to tradeoff the WAN usage and job performance.

Requirement 2: The algorithms should make adaptive scheduling decisions. As stated before, the bottleneck of job executions varies and is unpredictable. Thus the algorithms should make adaptive decisions when managing resources and scheduling tasks for jobs.

Algorithm 1 Af (applied by each job manager)

procedure Af($d(q-1)$, $a(q-1)$, $u(q-1)$)
  if q = 1 **then**
    $d(q) \leftarrow 1$
  else if $u(q-1) < \delta$ **and** *no waiting tasks* **then**
    $d(q) \leftarrow d(q-1)$ / ρ *//inefficient*
  else if $d(q-1) > a(q-1)$ **then**
    $d(q) \leftarrow d(q-1)$ *//efficient and deprived*
  else
    $d(q) \leftarrow d(q-1) \cdot \rho$ *//efficient and satisfied*
  end ifreturn $d(q)$
end procedure

C. Resource management using Af

Resources in a data center are scheduled by the job scheduler to sub-jobs between ***periods***; each period is of equal time length L. We denote the ***sub-job*** to the collection of tasks of a job that are executed in the same data center (and handled by the same JM). Fig. 7(b) shows an example of sub-job partition. For each sub-job J_i^j of job J_i, its job manager (pJM or sJM) applies Adaptive feedback (Algorithm 1) to determine the desire number of containers for next period $d(J_i^j, q)$ based on its last period desire $d(J_i^j, q-1)$, the last period allocation $a(J_i^j, q-1)$, the last period resource utilization $u(J_i^j, q-1)$ and waiting tasks. (We omit J_i^j in Algorithm 1 for brevity.) $u(J_i^j, q-1)$ corresponds to the average resource utilization in period $q-1$, and can be measured by the monitoring mechanism.

We classify the period $q-1$ as *satisfied* versus *deprived*. Af compares the job's allocation $a(J_i^j, q-1)$ with its desire $d(J_i^j, q-1)$. The period is satisfied if $a(J_i^j, q-1) = d(J_i^j, q-1)$, as the sub-job J_i^j acquires as many containers as it requests. Otherwise, the period is deprived. Af uses a parameter δ as well as the presence of waiting task information to classify a period

Algorithm 2 Parades (applied by each job manager)

```
1:  procedure onUpdate(n, δ, τ)
2:      For each task t_is, increase t_is.wait by the time since last event UPDATE; cont ← true
3:      if no waiting task then
4:          t = steal(n);
5:          tlist.add(t); n.free −= t.r; cont ← false;
6:      end if
7:      while n.free > 0 and cont do
8:          cont ← false;
9:          if there is a node-local task t_iu on n and n.free ≥ t_iu.r then
10:             t = t_iu
11:         else if there is a rack-local task t_iv on n and n.free ≥ t_iv.r and t_iv.wait ≥ τ · t_iv.p then
12:             t = t_iv
13:         else if there is a task t_iw with t_iw.wait ≥ 2τ · t_iw.p and n.free ≥ 1 − δ then
14:             t = t_iw
15:         end if
16:         tlist.add(t); n.free −= t.r; cont ← true;
17:     end while
18:     return tlist
19: end procedure
20: procedure onReceiveSteal(n)
21:     return onUpdate(n, δ, τ)
22: end procedure
23: procedure steal(n)
24:     for each job manager of the same job do
25:         tlist.add(sendSteal(n))
26:     end for
27:     return tlist
28: end procedure
```

as *efficient* versus *inefficient*. The period is inefficient if the utilization $u(J_i^j, q-1) < \delta$ *and* there is no waiting task in period $q-1$. Otherwise the period is efficient.

If the period is inefficient, Af decreases the desire by a factor ρ. If the period is efficient but deprived, it means that the sub-job efficiently used the resources it was allocated, but Af had requested more, so it maintains the same desire in period q. If the period is efficient and satisfied, the sub-job efficiently used the resources that Af requests. Af assumes that the sub-job can use more and increases its desire by a factor ρ. Af meets *Requirement 2* since it adaptively requests and releases resources due to the state of current system and the job.

D. Task assignment using Parades

When a new stage of a DAG job becomes available, the primary job manager initially decides the fraction of tasks to place on each data center to be proportional to the amount of input data on the data center.

Parades (**Para**meterized **de**lay scheduling with work **s**tealing) is applied by each JM after the initial assignment. Parades is based on framework of the original delay scheduling algorithm [40], but extends it from two perspectives. When a container updates status to its JM, the algorithm adds the waiting time for each waiting task of the sub-job since the last event *UPDATE* happened (Line 2), followed by the task assignment procedure. We modify the threshold for each task to be linearly dependent of its processing time p (which is *known*), under the intuition that "long" tasks can tolerate a longer waiting time to acquire their preferred resources. On the other hand, if there is no waiting task, the JM becomes a "thief" and tries to steal tasks from other "victim" JMs in the same job (Line 4). Each victim JM will handle this steal as a *UPDATE* event (Line 16).

Parades operates as follows in task assignment procedure: It first checks whether there is a node-local task waiting, which means the container *n* is on the same server as the task prefers. Assigning the task to its preferred server which containing its input data helps in reducing data transmission over the network. We use $n.free$ to denote the free resources on container *n*. Secondly, the algorithm would check whether there is a rack-local task for the *n*, as the container shares the same rack as the task's preferred server. If the task has waited for more than the threshold time ($\tau \cdot t_{ij}.p$), and the container has enough free resources, we assign the task to the container. Finally, when a task has waited for long enough time ($2\tau \cdot t_{il}.p$), and $n.free \geq 1-\delta$, we always allow the task could be assigned if possible.

See Table I for the involved notations in our algorithms. Parades satisfies *Requirement 2* because the thief JM can adaptively help (steal) other bottleneck (victim) JMs in runtime. The whole task assignment method meets *Requirement 1* for two reasons. Firstly the initial assignment pushes large number of tasks going to the data center with large input data, avoiding massive data crossing data center boundaries through WAN. Secondly a steal is successful only when tasks in victim JMs have waited for longer (enough) time than those in thief JMs. This may lead additional data going through WAN, but we believe it is *worthwhile* since moving tasks from bottleneck can effectively reduce job response time.

TABLE I
EXPLANATIONS OF NOTATIONS.

Notation	Explanation
$d(J_i^j, q)$	J_i^j's desire for period q
$a(J_i^j, q)$	J_i^j's allocation for period q
$u(J_i^j, q)$	J_i^j's resource utilization in period q
$n.free$	free resources on container n
$t_{is}.wait$	waiting time of task t_{is}
$t_{is}.r$	resource requirement of task t_{is}
$t_{is}.p$	processing time of task t_{is}
δ	the utilization threshold parameter
ρ	the resource adjustment parameter
τ	the task waiting time parameter

E. Analysis of Af + Parades

To prove the proposed algorithms guarantee efficient performance for online jobs, we settle the job scheduler employed in each data center as the fair scheduler [3], [7], perhaps the most widely used job scheduler in both industry and academia. We prove the following theorem about the competitive ratio of makespan. Specifically, we *extend* the very recent result [43] about the efficiency of jobs scheduled in a single data center.[1] Please see Appendix B for the proof sketch.

Theorem 1: When multiple geo-distributed DAG jobs arrive online and each data center applies fair job scheduler, the makespan of these jobs each applying Af + Parades, is $O(1)$-competitive.

V. IMPLEMENTATION

We implement HOUTU using Apache Spark [41], Hadoop YARN [34] and Apache Zookeeper [21] as building blocks. We make the following major changes in the original systems[2]:

Monitor mechanism: HOUTU estimates the dynamic resource utilization on each container by adding a resource monitor process (in nodeManager component of YARN). The monitor process reads resource usages (*e.g.*, CPU, memory) from OS counters and reports them to its JM. Each JM and its per-container monitors interact in an asynchronous manner to avoid overheads.

Parameterized delay scheduling: Based on the fact that tasks in a stage have similar resource requirements, we estimate the requirements using the measured statistics from the first few executions of tasks in a stage. We continue to refine these estimations as more tasks have been measured. Task processing time is estimated as the average processing time of all finished tasks in the same stage. We modify the original implementation of delay scheduling in Spark to take τ as a parameter read from the configuration file.

How the job managers coordinate with each other? As stated in §III, we use Zookeeper to synchronize JMs in the same job. Specifically, when the pJM determines the initial task assignment, it writes this information to taskMap (Figure 6(b)). JMs will notice this modification and begin their task assignment procedure using Parades (§IV-D). If a job manager successfully steals a task from another, it also needs to modify the corresponding item in taskMap. After a task completes, it reports to its job manager about the output location, who will then propagate the location information in partitionList among other job managers.

How a new job manager inherits the containers belonging to the failed one? We modify YARN master to allow to grant tokens to the new generated job manager with the same jobId as the failed one. Then, the new job manager could use these tokens to access the corresponding containers.

Af: HOUTU continuously (per second) measures the container utilizations in a sub-job J_i^j in a period q of length L, and calculates the average at the end of the period. HOUTU acquires the desire number of containers for the next period $d(q+1)$ by Af (§IV-C). If $d(q+1) \geq d(q)$, HOUTU directly updates the desire and pushes this new desire to the job scheduler. When $d(q+1) < d(q)$, the problem is involved, since HOUTU should decide *which* containers should be killed, and *when* the kill should be performed? HOUTU aggressively kills the several containers which firstly become free. We add this control information through the job manager in Spark to negotiate resources with YARN master.

VI. EXPERIMENTAL EVALUATION

In this section, we first present the methodology in conducting our experiments. Then, we show the efficient job performance HOUTU guarantees in both normal and artificially changeable environment, and analyze the monetary costs of HOUTU and other deployments when running the same workloads. Finally, we verify the ability of recovering of job manager failures in HOUTU and measure the overheads that it introduces in detail.

A. Methodology

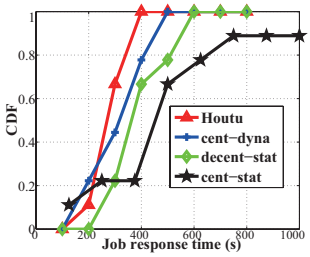
Testbed: We deploy HOUTU to 20 machines spread across four AliCloud regions as we show in §II-B. In each region, we start five machines of type `n4.xlarge` or `n1.large`, depending on their availability. Both types of instances have 4 CPU cores, 8GB RAM and run

[1] We extend the Pdelay algorithm in [43] with work stealing, which can *only* accelerate task assignment compared to Pdelay algorithm.

[2] The source code is available at https://github.com/DislabNJU/Houtu

Workloads	Input datasets		
	small	medium	large
WordCount	200 MB	1 GB	5 GB
TPC-H	--	1 GB	10 GB
Iterative ML	170 MB	1 GB	~ 3 GB
PageRank	150 MB	1 GB	~ 6 GB

Fig. 8. Input sizes for four workloads.



(a) CDF of job response time.

	Average JRT	Makespan
Houtu	290 s	387 s
cent-dyna	295 s	417 s
decent-stat	377 s	561 s
cent-stat	488 s	1109 s

(b) average job response time and makespan.

Fig. 9. Job performance in four deployments.

64-bit Ubuntu 16.04. In each region, we choose one On-demand instance as the master and four Spot instances as workers.

Workload: We use workloads for our evaluation including *WordCount*, *TPC-H benchmark*, *Iterative machine learning* and *PageRank*. For each workload, the variation in input sizes is based on real workloads from Yahoo! and Facebook in scale with our deployment (Fig. 8). For the job distribution, we set 46%, 40% and 14% of jobs are with small, medium and large input sizes respectively, which also conforms to reality [34]. For *TPC-H benchmark*, we place in each data center two tables, while for other three workloads, we evenly partition the input across four data centers.

Baselines: We evaluate the effectiveness of HOUTU by evaluating four main types of systems/deployments: (1) the centralized Spark on YARN system with built-in static resource scheduling (cent-stat); (2) the centralized Spark on YARN system with state-of-the-art dynamic resource scheduling (cent-dyna) [43]; (3) HOUTU, decentralized architecture with Af + Parades; (4) decentralized architecture with static resource scheduling (decent-stat).

Metrics: We use average job response time (JRT) and makespan to evaluate the effectiveness of jobs which arrive in an online manner. We also care about the monetary cost of running these jobs, compared with the deployment using total reliable (On-demand) instances. Finally, we are interested in JRT when facing failures.

B. Job performance

We use the workloads stated before, and set the job submission time following an exponential distribution with mean interval as 60 seconds. Fig. 9 shows the job performance in our four different deployments. First, we find that HOUTU has approximate performance compared with the centralized architecture with start-of-the-art dynamic scheduling mechanism. This approximation is due to that we allow job managers in a job to *share* resources across data centers by work stealing (Parades). Second, when compared with the decent-stat, HOUTU has 29% improvement in terms of average job response time, and 31% improvement in terms of makespan. This gain comes from the use of adaptively scheduling mechanism based on utilization feedback (Af).

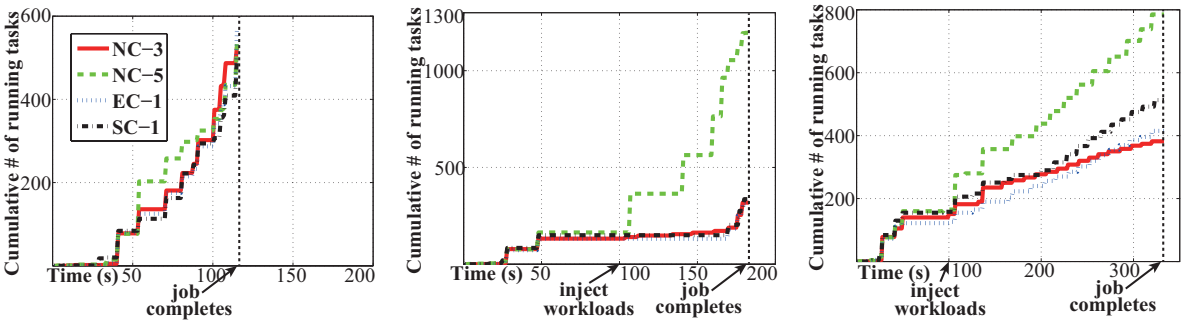
To further demonstrate that HOUTU guarantees efficient job performance in a changeable environment, we intentionally inject workloads to consume spare resources in data centers and see how a job reacts to this variation. Fig. 10 shows the cumulative running tasks of a job execution in different scenarios and mechanisms. In Fig. 10(a), a job executes normally and completes at time 115. While in Fig. 10(b) and Fig. 10(c), we inject workloads into three data centers NC-3, EC-1 and SC-1 to *use up almost all* spare resources in these data centers after 100 seconds of a job submission. Fig. 10(b) demonstrates that work stealing mechanism ensures that the job manager in NC-5 *gradually* steals tasks from the other resource-tense data centers as the new stages of the DAG job become available. However, without work stealing, the pJM assigns tasks only according to the data distribution (initial assignment), which then leads to that the sJMs in resource-tense data centers would *queue* the tasks to be executed. As shown in Fig. 10(c), the queueing delays the job. Job response times in the last scenarios are 183 and 333 seconds, respectively.

C. Cost analysis

In this subsection, we configure the other architectures with On-demand instances, while keep the HOUTU with Spot instances (except for the masters). We use the same workloads as in Fig. 9, and calculate the monetary costs in different deployments. Costs are divided into machine cost and data transfer cost across different data centers[3].

Fig. 11 shows two types of costs in different deployments normalized with the cost in cent-stat. First,

[3]In AliCloud [1], the price of data transfer across data centers is 0.13$/GB, while it is free to transfer data within a data center.

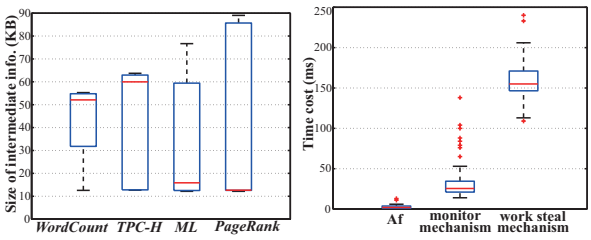


(a) The normal job execution in HOUTU (b) The normal job execution in HOUTU when we inject workloads (c) The job execution without work stealing mechanism when we inject workloads

Fig. 10. The cumulative running tasks of a job execution in different scenarios and mechanisms.

	machine cost	communication cost	total cost
Houtu	0.09	0.79	0.23
cent-dyna	0.37	0.84	0.46
decent-stat	0.50	0.77	0.55

Fig. 11. Monetary costs of different architectures normalized by the costs of cent-stat.



(a) Intermediate info. size of four workloads on *large* input. (b) Time cost of different mechanisms in HOUTU.

Fig. 13. Intermediate information size and time cost.

we observe HOUTU is very effective in reducing the machine cost of running geo-distributed jobs, which is 90% cheaper than the cost in cent-stat. The cost saving comes from the use of Spot instances, as well as the makespan reduction as optimized by HOUTU. Second, HOUTU has fewer data transfer compared with centralized architectures. This is because centralized architectures do not distinguish machines in different data centers; while HOUTU differentiates task assignment within a data center and between data centers. HOUTU saves about 20% communication cost compared with cent-stat. We also observe that HOUTU has a little higher WAN traffic compared to decent-stat, which is because the work stealing increases data transmission, but this highly valuable traffic significantly improves the job performance. In total, HOUTU reduces about three quarters of monetary cost of cent-stat.

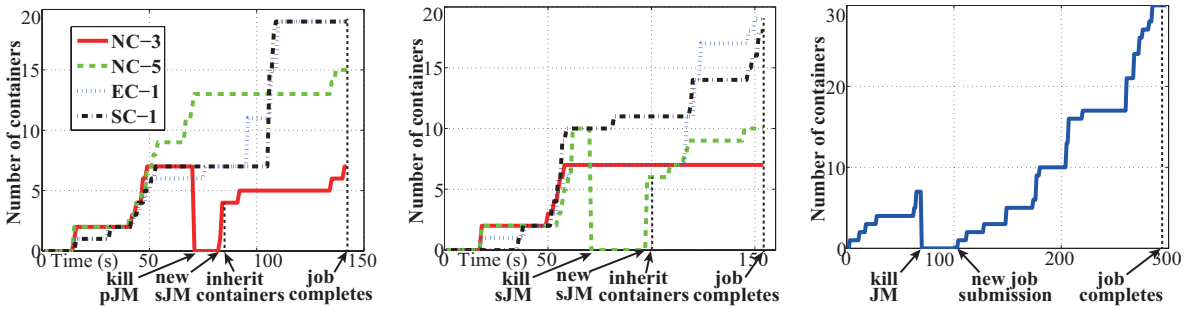
D. Failure recovery

HOUTU is designed to ensure that a job could recover from a failure due to the unpredictable termination of host. To understand the effectiveness of our proposed mechanism, we respectively run a job in HOUTU and cent-dyna, and we *manually* terminate the host (VM) where the JM resides at 70 seconds after its submission. We count the number of containers belonging to the job. Fig. 12 shows the process of the job when experiencing a JM failure. In Fig. 12(a), we kill the VM which hosts the pJM, and after 10 seconds we see a new sJM replaces the failed pJM. The sJM then inherits the old containers and continues its work. While in Fig. 12(b), we kill a sJM and see the similar process. The interval time is always lower than 20 seconds in our extensive experiments. The job response times in two scenarios are 147 seconds and 154 seconds, respectively. However, in the centralized architecture, the failure of a job manager leads to the resubmission of the job, which wastes the previous computations. The job response time is 299 seconds in the last case, which is significantly longer than the times in two executions in HOUTU.

E. Overhead

We measure overheads of HOUTU from two aspects.

First, we collect the intermediate information of jobs from four workloads on large input datasets, and measure their sizes during their executions. Fig. 13(a) plots the 25th percentile, median and 75th percentile sizes for each



(a) Recovery of a pJM failure in HOUTU (b) Recovery of a sJM failure in HOUTU (c) A job manager failure in cent-dyna

Fig. 12. The job failure recovery in HOUTU and the centralized architecture

workload in the corresponding box. We find the average sizes for the four workloads are 43.1 KB, 43.4 KB, 37.8 KB and 30.8 KB, respectively, which are small enough to leverage Zookeeper to keep them consistent.

Second, we measure the time costs of mechanisms that HOUTU introduces. For the Af overhead, it just maintains the update operation and incurs negligible costs. Compared to the default implementation in YARN, we add the monitoring mechanism in each container process, which has moderate overhead. As a job manager incurs transmission delay in work stealing, we find the average delay of the steal message transmissions is 163.5 ms across different system loads, which is also acceptable, as shown in Fig. 13(b).

VII. RELATED WORK

Wide-area data analytics: Prior work establishes the emerging problem of analyzing the globally-generated data in data analytics systems [18]–[20], [30], [36]–[38]. These works show promising WAN bandwidth reduction and job performance improvement. SWAG [20] adjusts the order of jobs across data centers to reduce job completion times. Iridium [30] optimizes data and task placement to reduce query response times and WAN usage. Clarinet [36] pushes wide-area network awareness to the query planner, and selects a query execution plan before the query begins. The proposed solutions work in the centralized architecture and assume the WAN bandwidth as constant, however, these may not conform to the practical scenario due to our argument in §II. Furthermore, we focus on the design of the decentralized geo-distributed data analytics architecture and requires no modification to the current job descriptions.

Scheduling in a single data analytics system: Data-locality is a primary goal when scheduling tasks within a job. Delay scheduling [40], Quincy [23] and Corral [25] try to improve the locality of individual tasks by scheduling them close to their input data. Fairness-quality tradeoff between multiple jobs is another goal. Carbyne [15] and Graphene [16] improve cluster utilizations and performances while allowing a little unfairness among jobs. Most of these systems rely on the priori knowledge of DAG job characteristics. Instead, we use utilization feedback to dynamically adjust scheduling decisions with only partial priori knowledge. Further, we extend this mechanism in the context of geo-distributed data centers and allow job managers to cooperate in scheduling tasks.

Fault-tolerance for jobs in data analytics: In current systems like MapReduce [13], Dryad [22] and Spark [41], each job manager tracks the execution time of every task, and reschedules a copy task when the execution time exceeding a threshold (straggler). At the level of jobs, the cluster (job scheduler) will resubmit a job when its reports are absent for a while. The resubmitted job starts its execution from scratch, wasting the previous computations. In the relevant grid computing, fault-tolerance of jobs is achieved by checkpointing, which is the collection of process context states [27], [29]. The process context states are stored periodically on a stable storage, which is not applicable in the data analytics systems due to the overhead for each job manager collecting the real-time task states and then persisting them. We include the output location for each task (partitionList) instead of its context state in its intermediate information, which is effective and incurs acceptable overheads as evidenced in our experiments.

VIII. CONCLUSION

We introduce HOUTU, a new data analytics system that is designed to support general analytics jobs on globally-generated data with respect to the regulatory

constraints. HOUTU provides a job manager for a job in each data center, ensuring the reliability of its execution. We present the strategy for each JM to independently manage resources under the help of utilization feedbacks instead of relying on complete priori knowledge of the job. We design the mechanism for JMs in a job to cooperatively assign tasks which allows JMs to share resources across different data centers with the data locality constraints. We experimentally and theoretically verify the job performance, and observe the effectiveness of cost reduction in HOUTU. We conclude that HOUTU is an regulation-abiding and efficient system to enable constrained globally-distributed analytics jobs.

REFERENCES

[1] Alibaba Cloud – Pricing. https://ecs-buy.aliyun.com/price.

[2] Amazon Web Services. Aws identity and access management (iam). https://aws.amazon.com/iam/.

[3] Apache YARN – Fair Scheduler. http://tinyurl.com/j9vzsl9.

[4] Cloud Services Pricing – Amazon Web Services (AWS). https://aws.amazon.com/pricing/.

[5] European Commission press release. Commission to pursue role as honest broker in future global negotiations on internet governance. https://tinyurl.com/k8xcvy4.

[6] Google Cloud Platform – Price List. https://tinyurl.com/y9nyq68e.

[7] Max-min fairness. https://tinyurl.com/krkdmho.

[8] Microsoft Azure – Pricing Overview. https://tinyurl.com/zk5kvla.

[9] Amazon. AWS global infrastructure. https://tinyurl.com/px6dzut.

[10] T. Brecht, X. Deng, and N. Gu. Competitive dynamic multiprocessor allocation for parallel applications. *Parallel Processing Letters*, 07(01), 1997.

[11] A. Burtsev, D. Johnson, J. Kunz, E. Eide, and J. Van der Merwe. Capnet: Security and least authority in a capability-enabled cloud. In *SoCC*, 2017.

[12] R. Buyya, J. Broberg, and A. M. Goscinski. *Cloud computing: Principles and paradigms*, chapter 24: Legal Issues in Cloud Computing. 2010.

[13] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[14] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM*, 2014.

[15] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *OSDI*, 2016.

[16] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *OSDI*, 2016.

[17] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven wan. In *SIGCOMM*, 2013.

[18] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu. Gaia: Geo-distributed machine learning approaching LAN speeds. In *NSDI*, 2017.

[19] C.-C. Hung, G. Ananthanarayanan, L. Golubchik, M. Yu, and M. Zhang. Wide-area analytics with multiple resources. In *EuroSys*, 2018.

[20] C.-C. Hung, L. Golubchik, and M. Yu. Scheduling jobs across geo-distributed datacenters. In *SoCC*, 2015.

[21] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.

[22] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[23] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[24] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. In *SIGCOMM*, 2013.

[25] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *SIGCOMM*, 2015.

[26] D. Kondo, B. Javadi, P. Malecot, F. Cappello, and D. P. Anderson. Cost-benefit analysis of cloud computing versus desktop grids. In *IPDPS*, 2009.

[27] H. Lee, K. Chung, S. Chin, J. Lee, D. Lee, S. Park, and H. Yu. A resource management and fault tolerance services in grid computing. *J. Parallel Distrib. Comput.*, 65(11), 2005.

[28] Microsoft. Azure regions. https://tinyurl.com/y98skbet.

[29] A. Nguyen-Tuong and A. S. Grimshaw. *Integrating fault-tolerance techniques in grid applications*. University of Virginia, 2000.

[30] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica. Low latency geo-distributed data analytics. In *SIGCOMM*, 2015.

[31] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman. Aggregation and degradation in jetstream: Streaming analytics in the wide area. In *NSDI*, 2014.

[32] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *CCS*, 2009.

[33] M. Rost and K. Bock. Privacy by design and the new protection goals. In *DuD*, 2017.

[34] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *SoCC*, 2013.

[35] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI*, 2016.

[36] R. Viswanathan, G. Ananthanarayanan, and A. Akella. CLARINET: Wan-aware optimization for analytics queries. In *OSDI*, 2016.

[37] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *NSDI*, 2015.

[38] H. Wang and B. Li. Lube: Mitigating bottlenecks in wide area data analytics. In *HotCloud*, 2017.

[39] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *SOSP*, 2013.

[40] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

[41] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[42] Y. Zhai, L. Yin, J. Chase, T. Ristenpart, and M. Swift. Cqstr: Securing cross-tenant applications with cloud containers. In *SoCC*, 2016.

[43] X. Zhang, Z. Qian, S. Zhang, X. Li, X. Wang, and S. Lu. COBRA: Toward provably efficient semi-clairvoyant scheduling in data analytics systems. In *INFOCOM*, 2018.

[44] A. C. Zhou, B. He, and C. Liu. Monetary cost optimizations for hosting workflow-as-a-service in iaas clouds. *IEEE Transactions on Cloud Computing*, 4(1), 2016.
[45] A. C. Zhou, S. Ibrahim, and B. He. On achieving efficient data transfer for graph processing in geo-distributed datacenters. In *ICDCS*, 2017.

APPENDIX A
PROBLEM FORMULATION

Suppose there is a set of jobs $\mathcal{J} = \{J_1, J_2, ..., J_{|\mathcal{J}|}\}$ to be scheduled on a set of containers $\mathcal{P} = \{P_1, P_2, ..., P_{|\mathcal{P}|}\}$ from all data centers. These containers are different since they reside in different servers (and different data centers) containing different input data for jobs. Time is discretized into ***scheduling periods*** of equal length L, where each period q includes the interval $[L \cdot q, L \cdot (q+1) - 1]$. L is a configurable system parameter.

We model a job J_i as a DAG. Each vertex of the DAG represents a task and each edge represents a dependency between the two tasks. Each task in a job prefers a unique subset of $\mathcal{P}$, as the containers in the subset store the input data for the task. For each task $t_{ij} \in J_i$, we denote by $t_{ij}.r$ to be the peak requirements. We assume $0 \leq t_{ij}.r \leq 1$, normalized by the container capacity. We also assume $t_{ij}.r \geq \theta$, where $\theta > 0$, *i.e.*, a task must consume some amount of resources. We associate $t_{ij}.p$ to be the processing time of task t_{ij}. Furthermore, the ***work*** of a job J_i is defined as $T_1(J_i) = \sum_{t_{ij} \in J_i} t_{ij}.r \cdot t_{ij}.p$. The release time $r(J_i)$ is the time at which the job J_i is submitted. A task is called in the *waiting* state when its predecessor tasks have all completed and itself has not been scheduled yet.

The ***sub-job*** J_i^j of J_i corresponds to a collection of tasks executing in the data center j. Each JM handles the task executions of a sub-job in the JM's data center. The JMs of a job are oblivious to the further characteristics of the unfolding DAG.

Definition 1: The **makespan** of a job set $\mathcal{J}$ is the time taken to complete all the jobs in $\mathcal{J}$, that is, $T(\mathcal{J}) = \max_{J_i \in \mathcal{J}} T(J_i)$, where $T(J_i)$ is the completion time of job J_i.

Definition 2: The **average response time** of a job set $\mathcal{J}$ is given by $\frac{1}{|\mathcal{J}|} \sum_{J_i \in \mathcal{J}} (T(J_i) - r(J_i))$.

The job scheduler of a data center and a job manager interact as follows. The job scheduler reallocates resources between scheduling periods. At the end of period $q-1$, the job manager of sub-job J_i^j determines its desire $d(J_i^j, q)$, which is the number of containers J_i^j wants for period q. Collecting the desires from all running sub-jobs, the job scheduler decides allocation $a(J_i^j, q)$ for each sub-job J_i^j (with $a(J_i^j, q) \leq d(J_i^j, q)$). Once a job is allocated containers, the job manager further schedules its tasks. And the allocation does not change during the period.

The problem: Given a job set $\mathcal{J}$ and container set from all data centers $\mathcal{P}$, we seek for a combination of a job scheduler (how to allocate resources to sub-jobs), and job managers within each job (how to request resources and how to assign tasks to the given resources), which minimizes makespan and average response time of $\mathcal{J}$, while satisfying the task locality preferences.

APPENDIX B
EFFICIENCY OF THE MAKESPAN

We settle the job scheduler to be fair scheduler. Once there is a free resource, the fair scheduler always allocates it to the job which currently occupies the fewest fraction of the cluster resources, unless the job's requests have been satisfied. We first state a lemma from [43] and then use it to prove the efficiency of makespan in the context of geo-distributed DAG jobs running in multiple data centers.

Lemma 2: [43] In a single data center with container set $\mathcal{P}$, which applies fair job scheduler, when DAG jobs $\mathcal{J}$ running in it each applying Adaptive feedback algorithm to request resources and parameterized delay scheduling to assign tasks, the makespan of these jobs

$$\begin{aligned} \mathrm{T}(\mathcal{J}) &\leq (\frac{2}{1-\delta} + \frac{1+\rho}{\delta} + \frac{2\tau}{\theta}) \frac{T_1(\mathcal{J})}{|\mathcal{P}|} \\ &+ L \log_\rho |\mathcal{P}| + 2L \ . \end{aligned}$$

Proof sketch of Theorem 1: Assume there are k data centers, the sub-job set executing in data center j is $\mathcal{J}^j$ and there are $|\mathcal{P}_j|$ containers in date center j. In the J_i example of Figure 7(b), $J_i^1 \in \mathcal{J}^1$, $J_i^2 \in \mathcal{J}^2$, and $J_i^3 \in \mathcal{J}^3$. Denote $c_i = \frac{1}{|\mathcal{P}_i|}(\frac{2}{1-\delta} + \frac{1+\rho}{\delta} + \frac{2\tau}{\theta})$ and $d_i = L \log_\rho |\mathcal{P}_i| + 2L$. By directly applying lemma 2, we have for each i,

$$\mathrm{T}(\mathcal{J}^i) \leq c_i \cdot T_1(\mathcal{J}^i) + d_i \ .$$

Sum them up, we have

$$\begin{aligned} \sum_{i=1}^{k} \mathrm{T}(\mathcal{J}^i) &\leq c_{max} \cdot \sum_{i=1}^{k} T_1(\mathcal{J}^i) + \sum_{i=1}^{k} d_i \\ &= c_{max} \cdot T_1(\mathcal{J}) + \sum_{i=1}^{k} d_i \\ &= c_{max} |\mathcal{P}| \cdot \frac{T_1(\mathcal{J})}{|\mathcal{P}|} + \sum_{i=1}^{k} d_i \ , \end{aligned}$$

in which c_{max} is the max of c_i and the first equality is due to the definition of work. According to the fact $\sum_{i=1}^{k} \mathrm{T}(\mathcal{J}^i) \geq \mathrm{T}(\mathcal{J})$, we have

$$\mathrm{T}(\mathcal{J}) \leq c_{max}|\mathcal{P}| \cdot \frac{T_1(\mathcal{J})}{|\mathcal{P}|} + \sum_{i=1}^{k} d_i \ .$$

Since $\frac{T_1(\mathcal{J})}{|\mathcal{P}|}$ is a lower bound of $\mathrm{T}^*(\mathcal{J})$ due to [10], and the number of total containers in all data centers $|\mathcal{P}|$ is constant once the system is well configured, we complete the proof of theorem 1.