5-2017

Towards distributed machine learning in shared clusters: A dynamically-partitioned approach

Peng SUN

Yonggang WEN

Nguyen Binh Duong TA
Singapore Management University, donta@smu.edu.sg

Shengen YAN

Towards Distributed Machine Learning in Shared Clusters: A Dynamically-Partitioned Approach

Peng Sun*, Yonggang Wen*, Ta Nguyen Binh Duong* and Shengen Yan†
* Nanyang Technological University, Singapore, † Sensetime Group Limited
{sunp0003, ygwen, donta}@ntu.edu.sg, yanshengen@sensetime.com

***Abstract*—Many cluster management systems (CMSs) have been proposed to share a single cluster with multiple distributed computing systems. However, none of the existing approaches can handle distributed machine learning (ML) workloads given the following criteria: high resource utilization, fair resource allocation and low sharing overhead. To solve this problem, we propose a new CMS named Dorm, incorporating a dynamically-partitioned cluster management mechanism and an utilization-fairness optimizer. Specifically, Dorm uses the container-based virtualization technique to partition a cluster, runs one application per partition, and can dynamically resize each partition at application runtime for resource efficiency and fairness. Each application directly launches its tasks on the assigned partition without petitioning for resources frequently, so Dorm imposes flat sharing overhead. Extensive performance evaluations showed that Dorm could simultaneously increase the resource utilization by a factor of up to 2.32, reduce the fairness loss by a factor of up to 1.52, and speed up popular distributed ML applications by a factor of up to 2.72, compared to existing approaches. Dorm's sharing overhead is less than 5% in most cases.**

***Index Terms*—Cluster Resource Management, Distributed Machine Learning, Fairness**

I. Introduction

A diverse array of distributed computing systems (DCSs) have emerged to handle various big data applications. Prominent examples include Hadoop and Spark. To offer better performance when training machine learning (ML) models, a lot of distributed ML systems have been proposed based on the ParameterServer (PS) framework, such as MxNet [1], MPI-Caffe [2], TensorFlow [3] and Petuum [4]. These systems could decompose an application into a set of small tasks and execute them on multiple nodes in parallel [5].

Many cluster management systems (CMSs) have been proposed to run multiple DCSs in the same cluster for two reasons. First, users can pick the best DCS for each application [6]. Second, cluster sharing could considerably improve the cluster resource utilization and application performance [7]. Existing CMSs can be classified into six categories based on their cluster management strategies. Specifically, Infrastructure-as-a-Service (IaaS) approaches (e.g., OpenStack [8]) can share clusters at the level of DCSs. For example, we can create a set of virtual machines (VMs) for Spark, and run all Spark applications in this virtual cluster. Monolithic, two-level, shared-state, fully-distributed and hybrid approaches can allocate cluster resources at the level of applications and tasks, such as Yarn [9], Mesos [6], Quasar [7], etc.

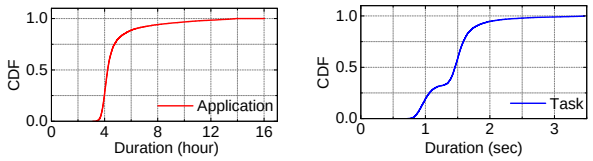


Fig. 1. CDF of distributed ML application and task duration.

In this work, we consider the problem of running multiple and diverse distributed ML workloads in a single cluster. For example, Sensetime Group Limited operates several clusters with thousands of CPUs/GPUs, and uses various distributed ML systems to train ML models on them. A workload analysis from one production cluster suggests that distributed ML applications usually have long application duration and very short task duration. Figure 1 shows that about 90% of distributed ML applications run more than 6 hours; and about 50% of distributed ML tasks use less than 1.5s.

However, none of the existing CMSs can efficiently handle distributed ML workloads in a shared cluster given three criteria: high resource utilization, low fairness loss[1] and low sharing overhead[2]. IaaS CMSs cannot handle distributed ML workloads at the level of DCSs, since popular distributed ML systems do not have multi-application support. For example, we need to manually allocate the resources of a TensorFlow virtual cluster to multiple applications. Monolithic, two-level, shared-state, fully-distributed and hybrid CMSs can only statically allocate resources to distributed ML applications, and do not allow them to dynamically scale up/down or scale out/in based on the global cluster state, resulting in low resource utilization and high fairness loss [6].

In this paper, we propose a new CMS named Dorm to handle multiple distributed ML workloads in a shared cluster with two techniques: a dynamically-partitioned cluster management mechanism and an utilization-fairness optimizer. Dorm uses the container-based virtualization technique to partition a cluster, and runs one application per partition. Each application places its tasks on the assigned partition without petitioning for resources, so Dorm imposes low sharing overhead. When detecting newly submitted or completed applications, Dorm

[1]Low fairness loss indicates that each applications could receive a fair share of resources. Its detail definition can be found in Section IV.

[2]Sharing overhead denotes the percentage of an application's additional running time imposed by a CMS.

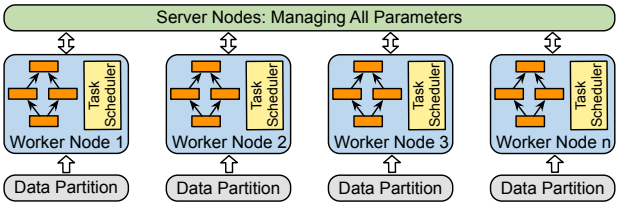


Fig. 2. The PS framework's architecture.

could adjust existing resource allocations to consistently keep high resource utilization and low fairness loss.

We implement Dorm using Docker and Cloud3dView [10], and integrate it with four widely used distributed ML systems: Petuum, MxNet, TensorFlow and MPI-Caffe. Extensive evaluations on a working testbed showed that Dorm could simultaneously improve the resource utilization by a factor of up to 2.32, reduce the fairness loss by a factor of up to 1.52, and speed up popular distributed ML applications by a factor of up to 2.72, compared to existing approaches. In most cases, Dorm could limit the sharing overhead within 5%.

II. Background and Related Work

In this section, we introduce distributed ML, review and analysis existing cluster management systems.

A. Distributed ML: A Primer

The goal of ML is to learn models from training datasets, and use them to make predictions on new data. To handle big training datasets and big models, many distributed ML systems have been proposed based on the PS framework. As shown in Figure 2, the PS framework can scale to large cluster deployment by having worker nodes performing data-parallel computation, and having server nodes maintaining globally shared *parameters* of ML models. Each worker node contains a TaskScheduler to place tasks on the local node based on a specific policy, such as Bulk Synchronous Parallel (BSP) or Stale Synchronous Parallel (SSP) [4].

B. Related Work: Cluster Management Systems

CMSs are designed to run multiple DCSs in a single cluster. As shown in Figure 3, existing CMSs can be classified into six categories based on their cluster management strategies. These approaches could perform resource allocation at three levels: DCS, application and task. Resource allocation refers to determining the amount of resources offered to applications, and selecting specific resources from servers to satisfy user-supplied placement preferences [7].

IaaS CMSs, such as OpenStack [8], use VMs to partition a cluster, run one DCS per partition, and let each DCS to manage and schedule submitted applications [11].

Monolithic CMSs, such as Yarn [9], Quasar [7] and Borg [12], use a centralized resource manager to perform resource allocation for all applications with cluster-wide visibility.

Two-level CMSs, such as Mesos [6], use a central cluster resource manager and application-specific schedulers to jointly perform resource allocation. The central manager gives each application a set of resource offers, and let the application-specific scheduler decide whether to accept them.

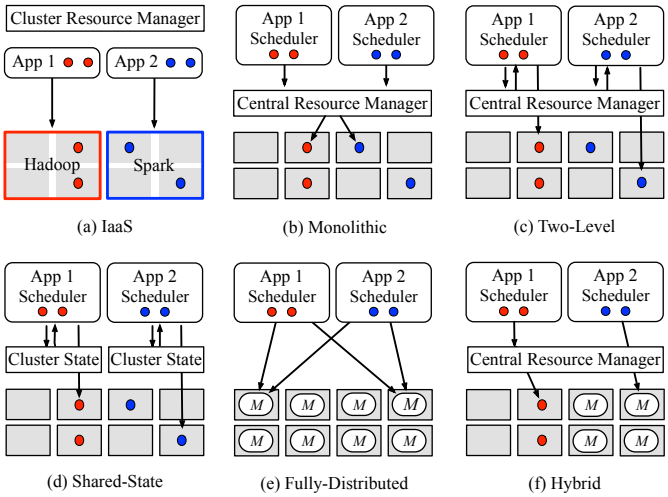


Fig. 3. Taxonomy of existing CMSs. Circles represent tasks; gray boxes represent cluster servers; and M denotes a distributed resource manager.

Shared-state CMSs let each application maintain a copy of the cluster state, and compete for resources using lock-free optimistic concurrency control, as in Omega [13] and Apollo [14]. These approaches could offer high resource allocation quality without strict fairness guarantees due to the lack of centralized resource management.

Fully-distributed CMSs, such as Sparrow [15], use many independent resource managers to serve applications' resource requests with local, partial and stale cluster state. This approach can achieve millisecond scheduling latency per request.

Hybrid CMSs combine distributed resource managers with a centralized cluster scheduler, as in Hawk [16] and Mercury [17]. Applications can obtain strong execution guarantees from the centralized scheduler, or trade strict guarantees for millisecond scheduling latency from distributed managers.

IaaS CMSs share cluster resources at the level of DCSs. This approach requires that DCSs could manage and schedule multiple applications. Monolithic, two-level, shared-state, fully-distributed and hybrid CMSs support both app-level and task-level resource allocation. In app-level mode, each application would reserve all allocated resources until completion. In task-level mode, applications would use acquired resources to run a single task, release them as soon as the task completes, and petition for new resources to launch uncompleted tasks.

C. Performance Analysis

Existing approaches cannot simultaneously achieve high resource utilization, low fairness loss and low sharing overhead when handling distributed ML workloads. IaaS CMSs cannot work in conjunction with popular distributed ML systems (e.g., TensorFlow), which have no multi-application support. In app-level sharing mode, monolithic, two-level, shared-state, fully-distributed and hybrid CMSs cannot dynamically adjust

existing resource allocations to consistently keep high resource utilization and low fairness loss. In task-level sharing mode, monolithic and two-level CMSs impose high sharing overhead, since each task must wait until receiving suitable resources. For example, in a 100-node Mesos cluster, our experiments showed that the average scheduling latency per task is about 430ms, which represents significant sharing overhead for short distributed ML tasks. Shared-state, fully-distributed and hybrid CMSs introduce concurrency control and distributed scheduling to reduce the sharing overhead at the cost of high fairness loss, due to the lack of centralized resource management.

In practices, existing CMSs could only statically allocate user-specified resources to distributed ML applications, as in TensorFlow-on-Mesos and MxNet-on-Yarn. When submitting a new application, users must manually specify its resource demands, including the number of worker nodes, and the amount of CPUs, GPUs and RAM per worker node.

III. Dorm: A Dynamically-Partitioned Approach

We propose a new CMS named Dorm to efficiently handle multiple and diverse distributed ML workloads in a single cluster using two techniques: a dynamically-partitioned cluster management mechanism and an utilization-fairness optimizer. In this section, we focus on the first technique.

A. System Architecture

Figure 4 shows Dorm's system architecture. Dorm is a type of the monolithic CMS, which contains a central DormMaster and a set of DormSlaves.

1) ***DormMaster****:* The DormMaster centrally manages all cluster resources, and exposes them to applications. It uses *containers*[3] to partition a cluster, and gives each application a partition. The utilization-fairness optimizer is a module of the DormMaster to make resource allocation decisions.

2) ***DormSlave****:* The DormSlave manages local resources of a cluster server. It reports the amount of available resources of a cluster server to the DormMaster, and uses *containers* to share a cluster server with multiple applications.

3) ***Application****:* Dorm is designed to host distributed ML applications. Since modern distributed ML systems usually use distributed scheduling mechanisms as shown in Section II, Dorm deploys a TaskExecutor and a TaskScheduler on each *container*. The TaskExecutor is the basic unit to execute tasks. The TaskScheduler is charge of placing tasks of an application on the local TaskExecutor.

4) ***Container****: Containers* of the same application would have uniform, constant resource demands for two reasons. First, distributed ML applications could balance the workloads across all TaskExecutors by equally partitioning the training datasets. Second, distributed ML applications usually use iterative methods to train models without changing resource demands during application runtime.

[3]The *container* is a logical bundle of resources on a server, for example ⟨2 CPUs, 1 GPU, 8GB RAM⟩.

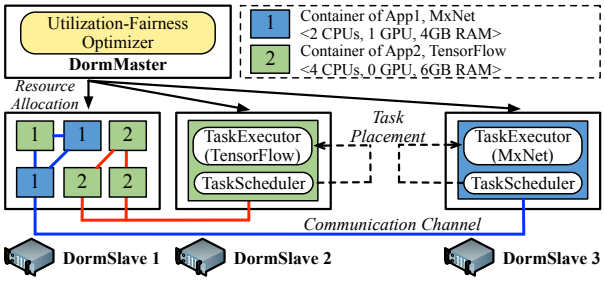


Fig. 4. Dorm's system architecture. In this example, a MxNet-based application and a TensorFlow-based application share a cluster with 3 servers.

B. Application Submission

To submit a new distributed ML application to Dorm, users need to provide a 6-tuple as follows:

$$(executor, \boldsymbol{d}, w, n_{max}, n_{min}, cmd),$$

where $executor$ is a string (e.g., "MxNet") to indicate the required computation engine; $\boldsymbol{d}$ is the resource demand vector (e.g., ⟨2 CPUs, 1 GPU, 8GB RAM⟩) per *container*; w is an integer to show this application's weight; n_{max} and n_{min} represent the maximum and minimum numbers of *containers* this application could have; cmd specifies the scripts used to start and resume this application.

C. Dynamically-Partitioned Resource Management

Dorm performs resource allocation in a dynamic manner at the level of applications. In a nutshell, it gives each application a partitioned cluster, and can dynamically resize each partition.

1) ***Making Resource Allocation Decisions****:* When detecting newly submitted or completed applications, the utilization-fairness optimizer determines new resource allocations for resource efficiency and fairness based on the algorithm detailed in Section IV.

2) ***Adjusting Existing Resource Allocations****:* Dorm could enforce new resource allocations by adjusting existing ones: creating and destroying *containers* on particular servers. However, popular distributed ML applications cannot automatically take advantage of newly acquired resources, or keep running with revoked resources. To address this problem, we propose a checkpoint-based resource adjustment protocol. Specifically, when adjusting an application's resources, Dorm would firstly save its state to a reliable storage system (e.g., the Lustre file system). Then, Dorm kills this application, and creates/destroys *containers* on corresponding servers. Finally, Dorm resumes the killed application from the saved state with new resource allocations. In this way, distributed ML applications can dynamically scale up or down without recomputing from the first iteration.

3) ***An Example****:* Figure 5 shows an example of how Dorm allocates resources to applications. In step (1), an user submits a new application to Dorm with following information:

$executor =$ "MPI-Caffe", $\boldsymbol{d} = \langle$1 CPU, 1 GPU, 8GB RAM$\rangle$, $w = 2, n_{max} = 5, n_{min} = 1, cmd =$ ["start.sh", "resume.sh"].

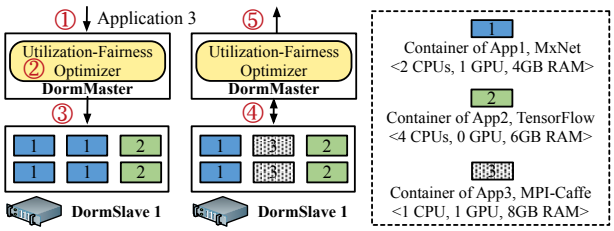


Fig. 5. An example to show how Dorm allocates resources to applications.

In step (2), the utilization-fairness optimizer determines that all applications should have 2 *containers* on DormSlave 1. In step (3), the DormMaster enforces new resource allocations by destroying 2 *containers* of App2 and creating 2 *containers* for APP3 on DormSlave 1. In this step, Dorm saves App2's state to a reliable storage system and kill it. In step (4), the DormMaster configures TaskExecutors and TaskSchedulers on new *containers*, starts APP3, and resumes APP2. In step (5), the DormMaster returns APP3's status to the user.

D. Task Placement

Dorm uses application-specific schedulers to place applications' tasks on assigned partitions. Since modern distributed ML systems use distributed scheduling mechanisms, Dorm deploys a TaskScheduler and a TaskExecutor per *container*. During application runtime, each TaskScheduler is in charge of placing tasks of an application on the local TaskExecutor. Therefore, application-specific schedulers would not request for resources to launch individual tasks, leading to low scheduling latency and low sharing overhead.

IV. Utilization-Fairness Optimizer

In this section, we show how the utilization-fairness optimizer makes resource allocation decisions. Table I shows used symbols and their definitions in this section.

A. Objectives

We consider a cluster with m types of hardware resources. When allocating cluster resources to the running application set $\mathcal{A}^t$ at time t, we aim to achieve high resource utilization and low fairness loss with low resource adjustment overhead.

*1) **Resource Utilization**:* The cluster's resource utilization is defined as the sum of all m types of hardware resources' utilization, which can be represented as follows:

$$\text{ResourceUtilization}(t) = \sum\nolimits_{k\in\mathcal{M}} u_k^t, \tag{1}$$

where $u_k^t = \sum_{i\in\mathcal{A}^t}\sum_{j\in\mathcal{B}} \frac{x_{i,j}^t d_{i,k}}{\sum_{h\in\mathcal{B}} c_{h,k}}$ denotes resource k's utilization at time t.

*2) **Fairness Loss**:* Fairness indicates that each application could receive a fair share of resources based on a particular fairness policy. In this paper, we use dominant resource fairness (DRF) [18] as the fairness policy. DRF seeks to maximize the minimum dominant share[4] across all applications. Let $\hat{s}_i^t$ denote application i's theoretical dominant share derived from DRF based on the algorithms proposed in [18]. Let s_i^t denote application i's actual dominant share. The cluster's fairness loss is defined as the sum of all applications' fairness loss, which can be represented as follows:

$$\text{FairnessLoss}(t) = \sum\nolimits_{i\in\mathcal{A}^t} l_i = \sum\nolimits_{i\in\mathcal{A}^t} \left|s_i^t - \hat{s}_i^t\right|, \tag{2}$$

where $s_i^t = \max_{k\in\mathcal{M}}(\frac{d_{i,k}\sum_{j\in\mathcal{B}} x_{i,j}^t}{\sum_{h\in\mathcal{B}} c_{h,k}})$.

*3) **Resource Adjustment Overhead**:* The cluster's resource adjustment overhead is measured by the number of affected applications, which would be killed and resumed, to enforce the newly computed resource allocations. Let r_i^t denote whether Dorm would adjust application i's resources:

$$r_i^t = \begin{cases} 0, & \text{if } x_{i,j}^{t-1} = x_{i,j}^t,\ \forall j \in \mathcal{B} \\ 1, & \text{if } x_{i,j}^{t-1} \neq x_{i,j}^t,\ \exists j \in \mathcal{B} \end{cases}. \tag{3}$$

If $r_i^t = 0$, Dorm would not create or destroy *containers* on any cluster servers for application i, and vice versa. It should be noted that the newly launched and completed applications at time t would not be considered as the affected applications due to resource adjustment. Therefore, the cluster's resource adjustment overhead can be represented as follows:

$$\text{ResourceAdjustmentOverhead}(t) = \sum\nolimits_{i\in\mathcal{A}^t\cap\mathcal{A}^{t-1}} r_i^t, \tag{4}$$

where $\mathcal{A}^t \cap \mathcal{A}^{t-1}$ is the set of applications running at both time $t-1$ and t.

B. Problem Formulation

Dorm determines the number of *containers* offered to applications, and the location of each *container*. We formulate this problem as a multi-objective optimization problem as follows:

$$\textbf{P1: } \max \left[\sum\nolimits_{i\in\mathcal{M}} u_i, -\sum\nolimits_{i\in\mathcal{A}^t} l_i, -\sum\nolimits_{i\in\mathcal{A}^t} r_i\right] \tag{5}$$

$$\text{s.t. } \sum\nolimits_{i\in\mathcal{A}^t} x_{i,j}^t d_{i,k} \leq c_{j,k},\ \ \forall k \in \mathcal{M}, \forall j \in \mathcal{B} \tag{6}$$

$$\sum\nolimits_{j\in\mathcal{B}} x_{i,j}^t \leq n_i^{max},\ \ \forall i \in \mathcal{A}^t \tag{7}$$

$$\sum\nolimits_{j\in\mathcal{B}} x_{i,j}^t \geq n_i^{min},\ \ \forall i \in \mathcal{A}^t \tag{8}$$

$$x_{i,j}^t \in \mathcal{Z}_0^+,\ \ \forall i \in \mathcal{A}^t, \forall j \in \mathcal{B} \tag{9}$$

TABLE I
Summary of Notations Used.

$x_{i,j}^t$	application i's *container* number on DormSlave j at time t.
l_i^t	application i's fairness loss at time t.
r_i^t	application i's resources are adjusted at time t.
u_i^t	resource i's utilization at time t
n_i^{max}	application i's maximum *container* number
n_i^{min}	application i's minimum *container* number
θ_1	threshold of fairness loss
θ_2	threshold of resource adjustment overhead
$\hat{s}_i^t$	application i's theoretical resource share based on DRF at time t
s_i^t	application i's actual resource share at time t
$d_{i,j}$	application i's resource demand on resource j
$c_{i,j}$	DormSlaves i's resource capacity on resource j
$\mathcal{B}$	set of DormSlaves, $\mathcal{B} = \{1, 2, \ldots, b\}$
$\mathcal{M}$	set of resource types, $\mathcal{M} = \{1, 2, \ldots, m\}$
$\mathcal{A}^t$	set of applications running at time t

[4]Dominant resource is the mostly heavily demanded resource required by an application, and dominant share is the share of the dominant resource.

Equation 5 is the objective function, which shows that we want to maximize resource utilization, minimize fairness loss and minimize resource adjustment overhead. We have several constraints. Equation 6 indicates that each cluster server cannot exceed its resource capacity. Equation 7 and 8 constraint the maximum and minimum numbers of *containers* an application can have. Equation 9 shows that $x_{i,j}^t$ is an integer variable.

We then transform **P1** into a MILP problem as follows:

$$\textbf{P2: } \max \sum_{k\in\mathcal{M}} \sum_{i\in\mathcal{A}^t} \sum_{j\in\mathcal{B}} \frac{x_{i,j}^t d_{i,k}}{\sum_{h\in\mathcal{B}} c_{h,k}} \tag{10}$$

$$\text{s.t. } l_i^t \geq s_i^t - \hat{s}_i^t, \quad \forall i \in \mathcal{A}^t \tag{11}$$

$$l_i^t \geq \hat{s}_i^t - s_i^t, \quad \forall i \in \mathcal{A}^t \tag{12}$$

$$Mr_i^t \geq x_{i,j}^{t-1} - x_{i,j}^t, \forall j \in \mathcal{B}, \forall i \in \mathcal{A}^t \cap \mathcal{A}^{t-1} \tag{13}$$

$$Mr_i^t \geq x_{i,j}^t - x_{i,j}^{t-1}, \forall j \in \mathcal{B}, \forall i \in \mathcal{A}^t \cap \mathcal{A}^{t-1} \tag{14}$$

$$\sum_{i\in\mathcal{A}_r^t} l_i^t \leq \lceil \theta_1 \times 2m \rceil, \tag{15}$$

$$\sum_{i\in\mathcal{A}_r^t} r_i^t \leq \lceil \theta_2 \times |\mathcal{A}^t \cap \mathcal{A}^{t-1}| \rceil, \tag{16}$$

$$l_i^t \in \mathcal{R}_0^+, \quad \forall i \in \mathcal{A}^t \tag{17}$$

$$r_i^t \in \{0,1\}, \quad \forall i \in \mathcal{A}^t \tag{18}$$

$$(6), (7), (8), (9).$$

In this formulation, we choose resource utilization as the objective to be maximized; fairness loss and resource adjustment overhead are constrained to be no greater than some given thresholds. Equation 11 and 12 are used to linearize l_i^t. Equation 13 and 14 are used to linearize r_i^t with a big number M. Equation 15 and 16 are the constraints for fairness loss and adjustment overhead with threshold θ_1 and θ_2, where $\theta_1 \in [0,1]$, $\theta_2 \in [0,1]$. We can see that **P2** is a typical MILP problem, which can be efficiently solved by standard MILP solves such as CPLEX. If there is no feasible solutions, Dorm would keep existing resource allocations until more running applications finish and release their resources.

V. Numerical Results and Analysis

In this section we investigate Dorm's performance using a testbed and popular distributed ML systems and applications.

A. Experiments' Parameters and Configurations

1) Testbed Setup: The testbed contains 21 computation servers (1 DormMaster and 20 DormSlaves) and 2 storage servers connected by 10Gbps Ethernet. All training datasets are stored on the two storage servers. The DormMaster manages 240 CPU cores, 5 GPUs and 2.5TB RAM in total.

2) Configurations: We use following thresholds for fairness loss and resource adjustment overhead in Dorm:

Dorm-1	$\theta_1 = 0.2$	$\theta_2 = 0.1$
Dorm-2	$\theta_1 = 0.1$	$\theta_2 = 0.2$
Dorm-3	$\theta_1 = 0.1$	$\theta_2 = 0.1$

3) Workloads: We generate an online workload based on the workload model of a production cluster in Sensetime. As shown in Table II, the workload comprises 50 applications, which train 7 ML models on public datasets. We randomly submit them to Dorm with a mean interval time of 20 minutes.

TABLE II
Synthetic workloads.

Dependent System	Training Datasets	Trained Model*	Resource Demand†	Weight	Max	Min	Num
MxNet	Criteo-Log	LR	2, 0, 8	1	32	1	20
TensorFlow	MovieLens	MF	2, 0, 6	2	32	1	20
MPI-Caffe	CIFAR-10	CaffeNet	4, 0, 6	4	8	1	6
MxNet	ImageNet	VGG-16	4, 1, 32	1	5	1	1
TensorFlow	ImageNet	GoogLeNet	6, 1, 16	1	5	1	1
Petuum	ImageNet	AlexNet	6, 1, 16	2	5	1	1
MPI-Caffe	ImageNet	ResNet-50	4, 1, 32	4	5	1	1

* LR: Logistic Regression; MF: Matrix Factorization.
† Number of CPUs, number of GPUs and RAM size (GB).

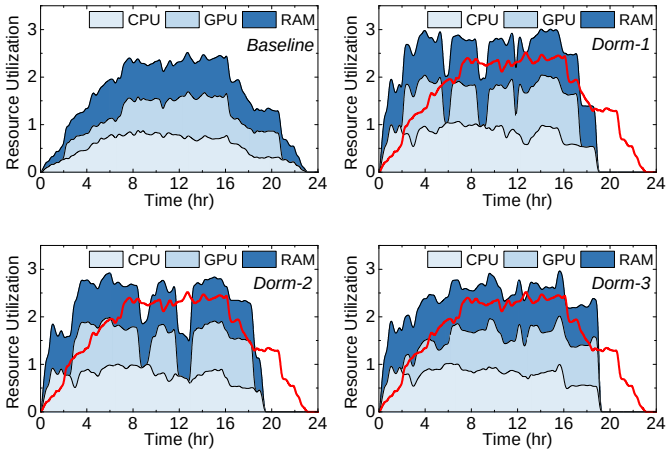


Fig. 6. Resource utilization of the testbed. The red line represents the overall resource utilization of the baseline system.

4) Baseline System: We use Swarm as the baseline system. In the experiments, Swarm would statically create 8, 8, 4, 2, 2, 2, 3 *containers* for the 7 types of applications in Table II.

B. Cluster Performance

1) Resource Utilization: Dorm could considerably improve the resource utilization, as shown in Figure 6. In the first 5 hours, the baseline system has quite low resource utilization (which is up to 1.8), since it can only handle the first 15 submitted applications based on their fixed resource requirements. In contrast, these applications could dynamically scale up to take advantage of more resources on Dorm. As a result, compared to the baseline system, Dorm-1, Dorm-2 and Dorm-3 can increase the resource utilization by a factor of 2.55, 2.46 and 2.32 on average in the first 5 hours, respectively.

2) Fairness Loss: As shown in Figure 7, Dorm limits the fairness loss within a threshold, and can tolerate higher fairness loss with a larger θ_1. Dorm-1 and Dorm-3, which set θ_1 to 0.2 and 0.1 with the same θ_2, can limit the fairness loss within 1.5 and 0.6, respectively. Though Dorm-1 provides higher resource utilization than Dorm-3, its fairness loss is up to 1.78 times higher than the baseline system. In contrast, Dorm-3 could reduce the fairness loss by a factor of 1.52 on average, compared to the baseline system.

3) Resource Adjustment Overhead: Figure 8 shows that Dorm can limit the resource adjustment overhead within a

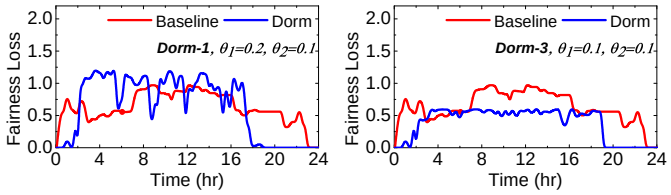


Fig. 7. Fairness loss of the testbed.

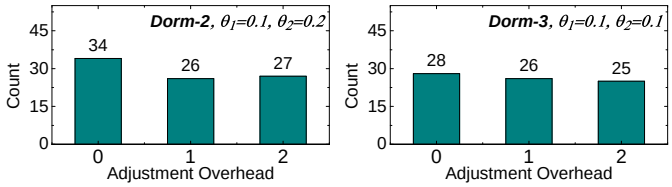


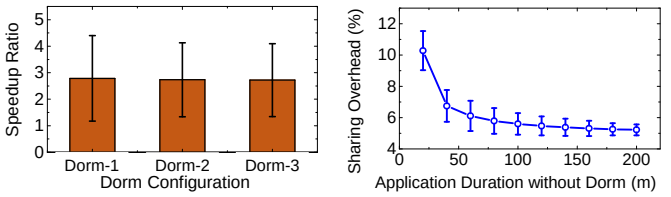
Fig. 8. Resource adjustment overhead of the testbed.

threshold, and can tolerate higher resource adjustment overhead with a larger θ_2. Dorm-2 and Dorm-3, which set θ_2 to 0.2 and 0.1 with the same θ_1, would kill and resume 2 applications at most per resource adjustment operation, and affect 80 and 76 applications in total in 24 hours, respectively.

4) Speedup Ratio: Distributed ML applications running on Dorm consistently perform better than those running on the baseline system. Figure 9(a) shows that Dorm-1, Dorm-2 and Dorm-3 can speed up distributed ML applications by a factor of 2.79, 2.73 and 2.72 on average, respectively.

5) Sharing Overhead: To measure Dorm's sharing overhead, we compare applications' performances in two cases. First, we set up a dedicated MxNet cluster on 10 worker nodes (each node has 4 CPUs and 16GB RAM), and run a set of applications on it. We then submit same applications to Dorm with the same amount of resources (i.e., $n_{max} = n_{min} = 10$, and each *container* has 4 CPUs and 16GB RAM). During the application running time, our tested MxNet-based applications are randomly killed and resumed 2 times on Dorm.

Dorm's sharing overhead is not significant for distributed ML applications. As shown in Figure 9(b), when the application duration is longer than 3 hours, Dorm would roughly increase the application duration by a factor of 1.05 (i.e., the sharing overhead of Dorm is about 5%). Compared to the performance gain, Dorm's sharing overhead is acceptable.



(a) Application Speedup Ratio (b) Sharing Overhead

Fig. 9. Application speedup ratio and Dorm's sharing overhead.

VI. Summary

We propose a novel cluster management system named Dorm to efficiently and fairly share a single cluster among distributed ML applications with low sharing overhead. To achieve this goal, Dorm employs a dynamically-partitioned sharing model and an utilization-fairness optimizer. We have implemented Dorm and enabled it to work with Petuum, MxNet, TensorFlow and MPI-Caffe. In the future, we plan to integrate it with more distributed ML systems and applications.

References

[1] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[2] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *ACM MM 14*. ACM, 2014, pp. 675–678.

[3] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *OSDI 16*. USENIX, Nov. 2016, pp. 265–283.

[4] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: a new platform for distributed machine learning on big data," *Big Data, IEEE Transactions on*, vol. 1, no. 2, pp. 49–67, 2015.

[5] H. Hu, Y. Wen, T.-S. Chua, and X. Li, "Toward scalable systems for big data analytics: A technology tutorial," *IEEE Access*, vol. 2, pp. 652–687, 2014.

[6] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI 11*, 2011, pp. 22–22.

[7] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.

[8] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "Openstack: Toward an open-source solution for cloud computing," *International Journal of Computer Applications*, vol. 55, no. 3, 2012.

[9] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache Hadoop Yarn: Yet another resource negotiator," in *SOCC 13*. ACM, 2013, p. 5.

[10] J. Yin, P. Sun, Y. Wen, H. Gong, M. Liu, X. Li, H. You, J. Gao, and C. Lin, "Cloud3dview: an interactive tool for cloud data center operations," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 499–500.

[11] Y. Jin, Y. Wen, Q. Chen, and Z. Zhu, "An empirical investigation of the impact of server virtualization on energy efficiency for green data center," *The Computer Journal*, vol. 56, no. 8, pp. 977–990, 2013.

[12] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *EuroSys 15*. ACM, 2015, p. 18.

[13] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *EuroSys 13*. ACM, 2013, pp. 351–364.

[14] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: Scalable and coordinated scheduling for cloud-scale computing," in *OSDI 14*, 2014, pp. 285–300.

[15] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *SOSP 13*. ACM, 2013, pp. 69–84.

[16] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid datacenter scheduling," in *USENIX ATC 15*, 2015, pp. 499–510.

[17] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga, "Mercury: Hybrid centralized and distributed scheduling in large shared clusters," in *USENIX ATC 15*, 2015, pp. 485–497.

[18] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types." in *NSDI 13*, vol. 11, 2011, pp. 24–24.