2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS) | 978-1-6654-4513-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICDCS51616.2021.00085

Harmony: A Scheduling Framework Optimized for Multiple Distributed Machine Learning Jobs

Woo-Yeon Lee[1], Yunseong Lee[2*], Won Wook Song[2], Youngseok Yang[2], Joo Yeon Kim[1], Byung-Gon Chun[2,3†]
[1]Samsung Research, [2]Seoul National University, [3]FriendliAI
{wooyeon.lee0, yunseong.lee0, wsong0512, johnyangk, jooykim00}@gmail.com, bgchun@snu.ac.kr

***Abstract*—We introduce Harmony, a new scheduling framework that executes multiple Parameter-Server ML training jobs together to improve cluster resource utilization. Harmony coordinates a fine-grained execution of co-located jobs with complementary resource usages to avoid contention and to efficiently share resources between the jobs. To resolve the memory pressure due to the increased number of simultaneous jobs, Harmony uses a data spill/reload mechanism optimized for multiple jobs with the iterative execution pattern. Our evaluation shows that Harmony improves cluster resource utilization by up to $1.65\times$, resulting in a reduction of the mean ML training job time by about 53%, and makespan, the total time to process all given jobs, by about 38%, compared to the traditional approaches that allocate dedicated resources to each job.**

I. Introduction

Machine learning (ML) training is one of the most popular data processing workloads in datacenters today. Especially, classical ML workloads are widely used in real-world production services [1]–[3] and are typically trained in large-scale shared CPU clusters [4]–[6].

Efficient resource scheduling among the ML jobs is key to improving cluster-wide performance. Existing scheduling systems for multiple ML jobs allocate a set of isolated resource units (e.g., containers, machines) that are obtained from resource managers, and a job exclusively runs on the allocated resource units [4]–[8]. For distributed ML training, the Parameter Server (PS) architecture is widely used in both research and industrial communities [2], [4], [6]–[13]. On the given set of resources, a PS-based job runs distributed *worker* and *server* tasks, where workers iteratively compute model gradients and synchronize progress by communicating through servers. However, existing scheduling systems result in an average utilization of around 50% of the assigned resources [9], [14]–[16], as a training job sequentially executes computation and communication steps, where each of step intensively uses a particular type of resource while leaving the others mostly idle.

In order to improve resource utilization, several works have proposed asynchronous training methods that disintegrate the sequential dependency of the computation and communication steps [17]–[19]. In such works, workers synchronize model parameters in the background while computing for gradients during the computation steps, making both the CPU and the network resources busy. However, breaking the sequential dependency often results in model inconsistency and computations based on stale models, and hence hinders model convergence [17], [20]–[22]. Although many works try to minimize occurrence of stale models in asynchronous training by constraining maximum staleness [18], [19], [23] or by differentiating the learning rate of the delayed updates [24], they have been able to reduce the side-effects but not completely resolve the issue, occasionally showing worse performance for complex models [21]. Due to such reasons, many works retain synchronous training to avoid staleness [20]–[22], [25].

A possible approach to resolve the under-utilization problem while keeping synchrony is to co-locate multiple jobs to share a pool of resources so that computation and communication can be interleaved. Nevertheless, naively co-locating jobs may result in multiple jobs contending for using the same type of resource simultaneously. This can result in an even slower job completion time than running each job alone (§V). In addition, ML training is memory-intensive [26], [27] and co-locating multiple jobs incurs even higher memory pressure, which results in job failures caused by out-of-memory errors, or slowdowns caused by garbage collection overheads, especially in managed runtimes like Java Virtual Machine environments.

To resolve these challenges, we introduce Harmony, a scheduling framework that co-locates and optimizes resource utilization among multiple ML training jobs, reducing the average job completion time (JCT) and makespan, the total time to complete all given jobs. Harmony exploits the pattern of ML job tasks that iteratively use different types of resources in each of their steps, to minimize resource contention. Specifically, Harmony first decomposes each job into fine-grained *subtasks*, each of which dominantly uses a single type of resource (e.g., network-subtasks, CPU-subtasks), then schedules the subtasks of co-located jobs in a pipelined manner, so that each subtask can fully utilize each type of resource without contending with the other subtasks in execution.

Moreover, as the performance of co-located jobs varies upon the set of jobs co-located and the number of machines allocated for the jobs, Harmony models the performance of co-located jobs with profiled metrics and runs a scheduling algorithm to make a decision towards higher resource utilization. As the pool of jobs changes with job arrivals and completions, Harmony dynamically reschedules jobs and resources to continuously find more efficient job groupings and resource allocation. To minimize overhead of continuous regroupings, we design our scheduling algorithm to minimize

*Yunseong Lee is currently at Qualcomm Technologies Inc.
† Corresponding author

2575-8411/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDCS51616.2021.00085

job movements and design our system to migrate jobs efficiently for multi-job situation.

Furthermore, under the higher memory pressure caused by the increased number of simultaneous jobs, Harmony prevents out-of-memory errors and garbage collection overheads with a data spill/reload mechanism optimized for multiple jobs with iterative execution patterns. Specifically, Harmony spills data that is not in active use to disk to relieve memory pressure. Since reloading data from disk is slow and has non-trivial overheads, we dynamically change the ratio of disk-side and memory-side data to balance overheads from memory pressure and disk read.

Our evaluation on 100 m4.2xlarge AWS EC2 instances shows that Harmony improves cluster resource utilization by up to 1.65× compared to traditional approaches that maintain dedicated allocation of resources. The increased resource utilization reduces average job training time by up to 53%, and makespan by up to 38%.

The rest of the paper is organized as follows: §II covers the background knowledge about the problem that we aim to solve, §III illustrates the overview of Harmony, §IV describes how Harmony divides jobs into subtasks and enables multiplexing of multiple jobs while mitigating memory pressures, §IV-B elaborates on how Harmony profiles and models jobs to predict and appropriately group jobs to co-locate on the provided nodes, §V presents the evaluation results and the comparison between Harmony and the baselines that represent existing systems, §VI discusses the limitations of Harmony and the future works, §VII covers related works, and §VIII concludes.

II. Background and Motivation

In this section, we first describe the parameter server (PS) architecture, a common framework designed for running large-scale distributed ML jobs, and how ML training jobs are scheduled to resources, to point out the inefficiency caused by idle resources in each execution step of existing PS-based systems. Then, we describe co-location of multiple jobs as a possible solution for the under-utilization of resources, and point out the challenges that arise with the approach.

A. Machine Learning in Parameter Servers

To facilitate large-scale ML training in distributed environments, systems designed with the PS architecture have been introduced [10]. The PS architecture mainly consists of *server*s that each maintains a partition of ML model parameters, and *worker*s that perform iterations of ML computations (e.g., deriving gradients) from each partition of input data. Workers synchronize with each other by communicating through servers via the push/pull APIs provided by the PS system. To highly utilize both CPU and network resources and to reduce the network overheads, workers and servers are usually located together [11], [12].

Figure 1 illustrates how a worker task performs in a training job. In a PS job, each *iteration*, or *mini-batch* processes a part of the input dataset, which altogether forms an *epoch*,

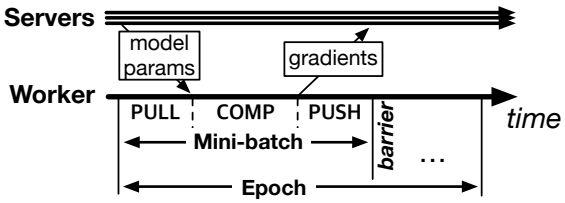


Fig. 1: The work-flow of a PS system. A worker w repeatedly performs *mini-batch*es, each composed of three steps (PULL-COMP-PUSH). Multiple mini-batches compose an *epoch*, which runs with the entire set of local input data.

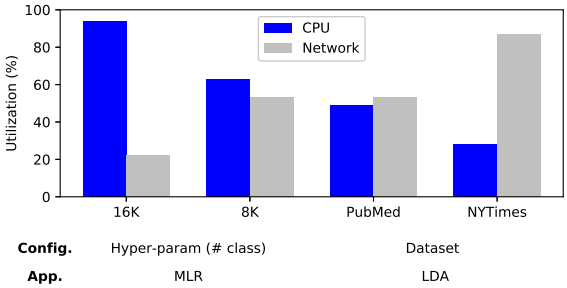
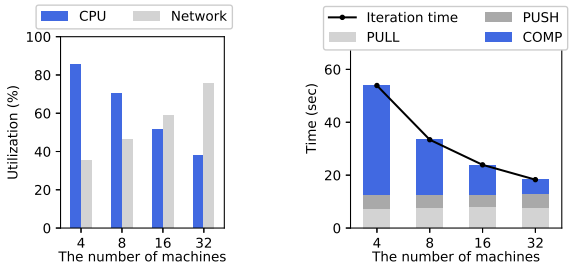


Fig. 2: ML training in PS fails to achieve high resource utilization, while showing different resource usage ratios with various workloads.

which describes a full scan of the training dataset. When an iteration begins, each worker first pulls the current model from servers (PULL), computes model gradients from the model and the assigned partition of input data (COMP), and pushes the gradients to servers to update the model (PUSH). The PS job repeats the process until sufficient epochs have been executed for the convergence of the model.

In an iteration, each step intensively uses a specific type of resource, while leaving the others mostly idle, resulting in an under-utilization of resources. In the COMP step, CPU and memory resources are intensively used, while network resources are heavily utilized in the PUSH and PULL steps. Figure 2 shows how CPU and network resources are underutilized while running different ML applications. In the experiment, we use multinomial logistic regression (MLR) and latent Dirichlet allocation (LDA) as workloads, which are widely used for classification and topic modeling. We run the experiment 10 times on 16 AWS m4.2xlarge EC2 instances using our PS system, which has comparable performance to an open-source PS system referenced in §V. In both applications, we can observe that the overall utilization rates stay rather indifferent, but also that the ratios of CPU and network utilization vary greatly.

Next, Figure 3 illustrates how the resource utilization changes with the number of machines allocated to the job. More number of machines naturally means that we could use higher degree of parallelism (DoP) using more CPU cores across multiple machines, leading to a shorter completion time, but also that the communication cost increases with more machines, leading to lower CPU resource utilization. On the other hand, less machines lead to less communication and thus higher utilization of CPU resources. Nevertheless, it



(a) Resource Utilization. **(b)** Job Completion Time.

Fig. 3: Running a job with different number of machines.

wastes network resources, which could be used to increase the parallelism of a job to shorten its execution time. In short, although increasing the number of machines results in better execution time and can adjust the ratio of CPU and network resource utilization rates, the resource under-utilization problem still remains as a challenge.

B. Co-location of Multiple Machine Learning Jobs

A possible approach in solving the inefficiency caused by idle resources of the different steps is to run multiple tasks of different jobs simultaneously. As the tasks that use different types of resources can run at the same time, we can expect the different types of resources to be utilized more intensively. Nevertheless, naively putting different jobs together does not solve the problem.

In Figure 4, we empirically show how co-located PS jobs may still lead to resource under-utilization. In addition to the MLR application used in Figure 2, we use non-negative matrix factorization (NMF) and lasso regression (Lasso) workloads, which are widely used for recommendation and regression problems, respectively. We compare the results when applications run on their own, and also when they run while they are co-located with others. When run on its own, each application shows varying levels of CPU and network resource utilization rates depending on the workload, as shown in the left half of Figure 4. Nevertheless, the overall utilization rates do not improve much even when co-located with other workloads, as shown in the right half of Figure 4. Co-location of two jobs in the example (e.g., NMF+Lasso, NMF+MLR) does not result in high utilization of both resources, but averages out the utilization for both of them to around 50%. Also, the standard error bars for co-location are much larger compared to when running each job on their own, which indicates that resource utilizations are more unpredictable. Co-locating all three jobs results in an out-of-memory error, as the sum of their memory use exceeds the amount of the total available memory. This indicates that higher utilization rates cannot be achieved by simply increasing the number of concurrent jobs.

As it can be seen from the observations, the resource under-utilization problem cannot be easily addressed with a black box approach, where jobs are naively co-located without being aware of the potential problems of the co-location. First, the root cause of under-utilization comes from the resource

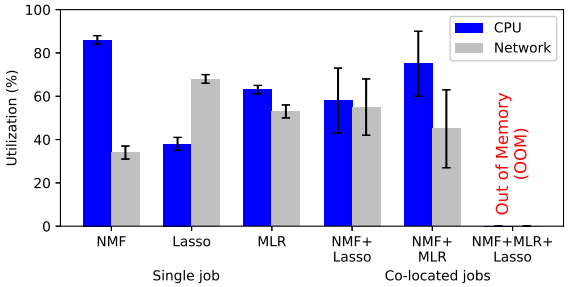


Fig. 4: Co-locating multiple PS jobs still fails to achieve high resource utilization rates.

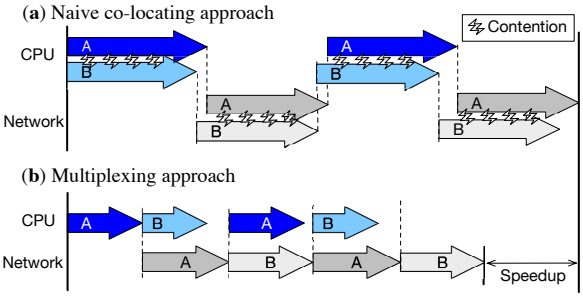


Fig. 5: Comparison of job scheduling approaches. The figure illustrates only two iterations of jobs for simplicity.

contentions that occur between the naively co-located jobs, as the tasks of different jobs that use the same type of resources compete with each other for the specific resource, as illustrated in Figure 5a. Such resource contention results in a lagged and unpredictable completion of each step of the job, and leaves big portions of resources idle. Second, the performance of co-located jobs is heavily dependent on the type of the co-located jobs. When grouping jobs together, one must carefully consider the characteristics and the complementary effects of their co-location, as otherwise it would lead to an imbalanced utilization of resources or even higher resource contention problems. Third, memory pressure from co-located ML jobs may result in job slowdown by GC overheads or job failures by OOM error. For performance reason, input data is often maintained in workers' memory because during training iterations workers repeatedly access the input data. Also model data is maintained in servers' memory to respond immediately for arbitrary accesses from workers. In addition, each training step consumes additional memory resources to generate intermediate results.

Therefore, in order to achieve higher resource utilizations, it is crucial to combine and execute tasks in a coordinated way, with the knowledge about resource use of each of the different task steps of different jobs. In the following section, we describe our solution to resolve these challenges and achieve efficient utilization of resources for co-location of PS ML jobs.

III. Harmony Overview

We introduce Harmony, a new scheduling framework that embodies our approach, which co-locates jobs with comple-

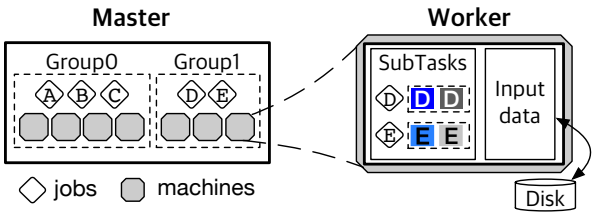


Fig. 6: Harmony scheduling overview. The master schedules jobs by grouping them and allocating resources, determined by the runtime metrics from workers. Workers schedule and execute subtasks, while dynamically reloading input data from disks in the background.

mentary resource use with each other and multiplexes their tasks to harmoniously share resources. To enable this, we provide three key techniques. First, we execute and control jobs with fine-grained scheduling unit called *subtask*s, each of which uses a specific type of resources as illustrated in Figure 5b. With subtasks, we can prevent the resource contention (e.g., CPU, network) with fine-grained management of the resource usage pattern during the execution co-located jobs.

Second, we co-locate jobs with complementary resource usage patterns to maximize the effect of job multiplexing. To solve this scheduling problem, we first model the performance of co-located jobs with the metrics collected during runtime. The subtask-based execution makes the performance predictable, and enables performance modeling. Based on the performance model, we devise a scalable scheduling algorithm that chooses the option for higher resource utilization, as well as for shorter execution times. In addition, to deal with the changing pool of jobs, we design a system and a scheduling algorithm to dynamically regroup jobs and to reallocate resources to them.

Lastly, we only maintain the input data of the subtasks in action in memory, while spilling the input data of other jobs on disk. This way, Harmony successfully relieves the memory pressure, by letting the jobs use memory resource in turns. However, as putting too much data on disk may lead to an increased latency for data loading due to a shortage of disk bandwidth, we dynamically balance the amount of input data in memory and disk. Also, we support similar mechanisms for the model data when the input data spill is not enough for mitigating the memory pressure.

Harmony provides a runtime to execute jobs, consisting of a *master*, with multiple *servers* and *workers*, as depicted in Figure 6. The master serves as a center for collecting metrics, grouping jobs into job groups, and scheduling them across available machines. Once a job is submitted, its worker and server code with its arguments are sent to the master, and the job is enqueued to the job queue with a `waiting` state. When the job is picked up, it gets naively assigned to a group and executed on the group's set of machines to be profiled. The master triggers the appropriate workers to load the input data, and servers to initialize their model parameters. Once they are set up, the master distributes its subtasks across workers, and the job enters the `profiling` state and the `profiled` and `running` state afterwards.

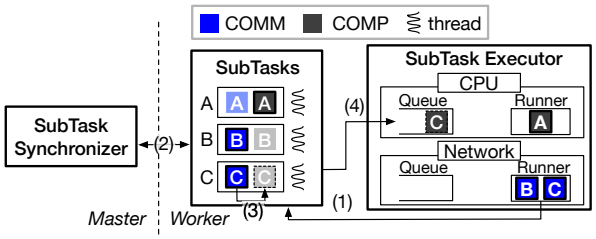


Fig. 7: Scheduling and execution of jobs A, B, and C, where A is at the COMP subtask, and B and C are at the COMM subtask.

Workers continuously collect runtime metrics during the execution to keep the master and job scheduler updated with the profiled metrics. Based on the profiled metrics, the job is assigned to a job group by the job scheduler, through the job scheduling algorithm described in §IV-B, and gets `paused` or migrated to the machine allocated for the optimized job group, with techniques that minimize the overhead on the progress of the jobs in execution. In the end, jobs are grouped into appropriate job groups, and each job group gets executed on the allocated machines until the convergence of the model (`finished`).

IV. Multiplexing ML Jobs

In this section, we first describe how Harmony executes multiple tasks in each worker with minimal contention using subtasks. We then describe scheduling for grouping jobs with complementary resource usage patterns, built upon the subtask-based execution model. Lastly, we describe our dynamic data reloading technique for relieving memory pressure caused by the multiple co-located jobs.

A. Fine-grained Execution with Subtasks

To minimize the resource contention between jobs, we decompose long-running worker tasks into smaller *subtasks*, each of which uses a single dominant type of a resource. In our context, COMP subtasks use CPU resources while PULL and PUSH subtasks use network resources. For the ease of representation, we call the network-intensive PULL and PUSH subtasks as COMM subtasks. The COMP and COMM subtasks from multiple different jobs can be coordinated so that only a single subtask can run at a time for a specific type of resources and the subtasks that require different types of resources simultaneously run together, utilizing the available resources.

Figure 7 illustrates how subtasks are scheduled and executed in a pipelined manner on Harmony. On the left, the *subtask synchronizer* in the master manages the state of the distributed job subtasks across multiple workers, to synchronize the overall progress of the job. On the right, in the worker, the *subtask*s get enqueued to the CPU or the network queue respectively, by the threads that run the subtasks for each job. The *subtask executor*s of the workers run the queued subtasks in the provided order. In a subtask executor, a single CPU subtask is executed at a time as a single CPU subtask

usually uses almost all of the provided CPU resources. On the other hand, as network subtasks often show asynchronous behaviors and cause idle network resources during the time that it takes for the servers to handle the pull/push requests, a single network subtask may not fully utilize the given network resources. To solve this, we schedule a secondary network subtask, while yielding the network resources to the primary network subtask whenever a contention occurs.

Subtask scheduling and execution are illustrated by an example shown in Figure 7(1-4). In the example, when a single COMM subtask of job C completes its execution (1), the *SubTask Synchronizer* checks the completeness of the other COMM subtasks of the other workers to synchronize the progress of the job (2). Then, when all distributed COMM subtasks of job C are complete, the COMP subtask of C is enqueued to the CPU queue (3-4), to be executed after the COMP subtask of A, which is already in execution. The executions of the subtasks occur in a similar manner for all other types of subtasks for each of the jobs.

Harmony does not require users to write their code with subtasks. Decomposing a worker task into subtasks can be done internally by the system, because model synchronization step is done by explicitly calling PS push/pull interfaces. Harmony naturally treats PS push/pull methods as COMM subtasks and the remainder parts of worker task as COMP subtasks. For better separation of resource use, we modify push/pull methods to minimize its CPU consumptions by performing data (de)serialization outside of COMM subtask.

In the following section, we describe higher-level scheduling problem of determining which jobs to co-locate and how many number of machines to allocate to co-located jobs.

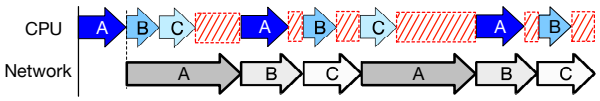
B. Dynamic Grouping of Jobs

Although now we have the techniques to run multiple co-located ML jobs without contention, the performance varies greatly based on which jobs are co-located together. Thus, it is crucial to co-locate jobs with complementary resource usages together.

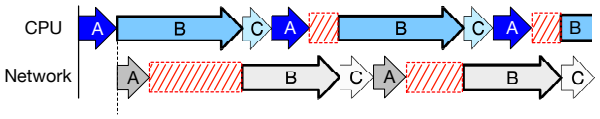
Harmony divides jobs into groups, where each group means a set of jobs to be co-located, and allocates a set of machine resources to each group. We call the group of co-located jobs as a *job group*, and a full iteration of the job group as a *group iteration*. A job group should have the balanced use of resources and the allocated resources should be kept busy during group iterations. To achieve this goal, Harmony makes a scheduling decision using runtime metrics and the performance model.

In this section, we describe how Harmony collects runtime metrics (§IV-B1) and models performance of co-located jobs based on the collected metrics (§IV-B2), and finally schedules jobs and machine resources (§IV-B3) and performs regrouping during runtime (§IV-B4).

1) Profiling: Fine-grained subtasks enable us to manage iterative ML jobs to run with smaller resource contention between their tasks, which makes it much easier to predict the performance of future iterations. ML jobs on Harmony show



(a) Resource-bound case (network-bound case): Sum of network-subtasks is longer than CPU-subtasks, leaving CPU resources idle.



(b) Job-bound case: Job B is too large compared to the other jobs. Both CPU and network resources are left idle.

Fig. 8: Problematic cases of unbalanced co-located jobs. Subtasks with bold lines incur under-utilization of resources. Red hatched boxes represent idle resources.

stable performance with reduced resource contention and thus the profiled metrics of subtasks can be meaningfully reused, while being updated using moving averages.

Harmony monitors each job j in each group g and collects runtime metrics which consists of the average execution times of CPU and Network subtasks and the number of machines allocated to the group $(T_{cpu_j}, T_{net_j}, m_g)$. The scheduler profiles the job in background, deploying to a job group with the smallest number of machines or a job group that is already profiling an another new job, to minimize the potential degradation of resource utilization.

2) Performance Modeling: When predicting group iteration time using the collected metrics, Harmony considers several cases of non-uniform resource use of jobs. Figure 8 shows two cases where naive subtask scheduling can be problematic. First, Figure 8a presents a resource-bound case, in which jobs in a job group are bounded by a certain type of resources due to imbalanced resource use, leaving the other type of resources idle. Second, Figure 8b shows a job-bound case, where a certain job has a much longer job iteration time compared to other jobs.

From the observations, we derive the equation for group iteration time $T_{g\_itr_g}$, which is the time for all jobs j of a group g to finish an iteration, with three terms: the maximum of job iteration times for the job-bound case, and the sum of COMP or COMM subtask times of the grouped jobs for the resource-bound case, as follows:

$$T_{g\_itr_g} = \max\left(\sum_{j\in g} T_{cpu_j}, \sum_{j\in g} T_{net_j}, \max_{j\in g} T_{j\_itr_j}\right) \quad (1)$$

The time for a COMP subtask T_{cpu_j} can be controlled by increasing the degree of parallelism (DoP), since each COMP subtask processes a smaller portion of input data with higher DoP, while COMM subtasks using network resources remain rather indifferent. Thus, T_{cpu_j} of a job j can be expressed with respect to the group DoP m_g of the job group g as follows:

$$T_{cpu_j} \propto \frac{1}{m_g}(j \in g) \quad (2)$$

This implies that manipulating the group DoP m_g of the job group may have an effect on $T_{g\_itr_g}$ according to Eq.1.

The utilization of CPU and network resources can be expressed as the percentage of the time spent by the subtasks for each type of resource, out of the group iteration time derived above ($T_{g\_itr_g}$). The utilization rates of CPU and network can be thus expressed as a two-dimensional vector as follows:

$$U(g) = \begin{bmatrix} U_{cpu_g} & U_{net_g} \end{bmatrix} = \begin{bmatrix} \dfrac{\sum_{j \in g} T_{cpu_j}}{T_{g\_itr_g}} & \dfrac{\sum_{j \in g} T_{net_j}}{T_{g\_itr_g}} \end{bmatrix} \quad (3)$$

If the job group is CPU-bound, then the CPU utilization rate becomes 1, and the same can be inferred for the network. In the job-bound case, the denominator ($T_{g\_itr_g}$) is larger than both the sum of CPU subtasks and network subtasks in the job group, leaving both type of resources partially idle.

We define the resource utilization of an entire cluster U as the weighted average of the utilization rates of all job groups, where G is the set of job groups:

$$U = \begin{bmatrix} U_{cpu} & U_{net} \end{bmatrix} = \frac{\sum_{g \in G} (m_g \times U(g))}{\sum_{g \in G} m_g} \quad (4)$$

Harmony constantly seeks for higher resource utilization U, and when it detects a potential improvement, it dynamically updates the jobs, job groups, and the allocated machines to increase efficiency.

There are a few other things that we consider with our performance model. First, we prefer fitting a smaller number of jobs in a job group for shorter JCTs and lower memory pressure. Second, CPU utilization rates are treated more importantly than the network utilization in our model, since CPU resources directly contribute to the job progress, whereas network resources is for communication.

3) Grouping Jobs and Allocating Machines: Based on the model, Harmony makes a scheduling decision that groups jobs and allocates machines to each job group. However, the scheduling problem is too complex with exponential time complexity and further Harmony requires the continuous scheduling corresponding to the changing pool of jobs. To be practical, we use heuristics that roughly determine initial values and do fine-tuning, which we show the scalability in §V-F.

Our scheduling algorithm (Algorithm 1) observes all jobs that are profiled and in the state of `running`, `paused`, or `profiled` (L2). While incrementing the number of jobs to consider, starting from a single job, Harmony tries to find the set of job groups G with better resource utilization, considering how to group them together and how to allocate resources to them (L4-13). Harmony first determines the number of groups n_G^*, which determines the DoP that balances the CPU and network usage of the jobs the most, assuming that all groups have an equal number of machines and thus the same DoP for all of the jobs (L6). Here, since

Algorithm 1: Job scheduling algorithm.

input : $J_{profiled}$: list of profiled jobs,
J_{paused}: list of paused jobs,
$J_{running}$: list of running jobs,
M: set of machines
output: G: grouping that maximizes utilization.

```
1  Function schedule(J_profiled, J_paused, J_running, M):
2      J_to_sched ← J_profiled ∪ J_paused ∪ J_running
3      U_max ← 0
4      for n_j ← 2 to |J_to_sched| do
5          J_to_group ← J_to_sched[0 : n_j − 1]
6          n_G* ← argmin_{n_G} Σ_{j∈J_to_group} |T_cpu_j(n_G) − T_net_j|
7          G_J ← assignJobs(J_to_group, n_G*)
8          G_M ← allocateMachines(G_J, M, n_G*)
9          G ← (G_J, G_M)
10         if U(G) > U_max then
11             U_max ← U(G)
12         else
13             break
14     return G
```

we assume that the DoP is equal among the job groups, $m_g \propto \frac{1}{n_G}$ and thus $T_{cpu} \propto n_G$ ($\because Eq.2$). Then, with the number of job groups decided, Harmony performs a grouping algorithm (L7), and allocates machines to the job groups (L8). With this information, Harmony computes the potential resource utilization $U(G)$, and continues with the loop if it sees potential improvement in the overall utilization (L10). Once it sees no more improvement with the increasing set of jobs, Harmony stops and runs the jobs with the optimized set of job groups (L12-14).

The grouping algorithm (L7) assigns jobs J evenly into a given number of groups n_G^*. In order to prevent job-bound cases, we place jobs with similar iteration times together as much as possible. For example, if large jobs are spread around each of the job groups, it would result in a longer average group iteration time, so we try to keep the large ones together. In order to do this, the scheduler first sorts jobs by their job iteration time $T_{j\_itr_j}$. The scheduler then fills job groups one by one with jobs from the sorted list in a greedy manner to balance resource use. Lastly, the algorithm fine-tunes the result by swapping jobs between the groups. It first picks the most imbalanced group, and finds the group that has the most complementary resource use. Then, it finds the tuple of jobs from each of the groups that would minimize the "resource-imbalance" for both of the groups, and swaps the two jobs between the groups. The fine-tuning repeats until there are no possible swap cases.

After the job assignment, we distribute the machines to the job groups (L8) to balance the computation and communication in each job group. First, the algorithm allocates one machine for every job group. The algorithm then repeats a step of allocating one machine to a group that needs additional machines the most. Those groups that need machines are the most computation-intensive ones, as having more machines would reduce the computation cost in an iteration ($\because Eq.2$), reducing the CPU-bound cases (Eq.1).

4) Dynamic Job Regrouping: When (1) a new job is submitted or (2) a job completes execution, scheduling has to be triggered, in order to look for the set of job groups that best fit the newly updated set of jobs. Since regrouping may cause extra overhead, we minimize the number of jobs participating in regrouping using the following regrouping algorithm. (1) When a new job arrives, the scheduler first performs profiling as described in §IV-B1. After profiling, the scheduler handles the job only when there is no other `profiled/paused` jobs, because existence of those jobs means that Harmony already satisfies with the currently `running` jobs. The scheduler handles the job by adding it to a proper group that maximizes U or let it wait if it does not improve U. (2) When one of existing jobs finishes, Harmony needs to repair a group of the finished job to be computation-communication balanced again. The scheduler searches for a similar job in terms of iteration time and comp/comm ratio among `profiled/paused` jobs to replace the finished job. When failing to find a similar job, the scheduler searches for a bunch of jobs with equivalent characteristics, whose the sum of iteration times and the ratio of respective sum of computation and communication times are similar to the finished job. We judge that jobs are similar when the difference of statistics is within 5%, which is an acceptable error as we shown in §V-E. If the scheduler fails to replace the finished job with `profiled/paused` jobs, the scheduler involves other job groups in regrouping, using the main scheduling algorithm (Algorithm 1). The scheduler calls `schedule` function altering $J_{running}$ with jobs in selected job groups. At first, the scheduler selects a group with the smallest number of jobs in addition to the group that the finished job belonged to. Then it changes the job group or adds more job groups, in the way of incrementally increasing the number of jobs that participate in regrouping. After finishing all possible combinations of job groups, it compares their predicted performance and selects the grouping decision with smaller number of jobs, if the performance improvement of decisions with more number of jobs is less than 5% compared to the decision with smaller number of job groups. In the same context, Harmony does not perform regrouping when the expected benefit is less than 5% of U.

To apply the new grouping decision by the scheduler, Harmony migrates running jobs between job groups and machines, also enabling reallocation of machines between the groups. During the migration of a job, the master simply pauses the job and executes the other co-located jobs in the meanwhile, keeping the resources busy. Harmony migrates only the stateful model parameters, which are trickier to handle, and simply reloads the immutable input data. In the case of the local states of subtasks (e.g., pulled model parameters, computed gradients), we simply perform the migration at the end of the iteration (i.e., after PUSH subtask). When temporarily pausing a running job during runtime, Harmony waits until ongoing iteration ends, stops the subtasks of the job, and checkpoints the model parameters on disk. Whenever it decides to resume the job, Harmony reloads the input data, restores the model parameters from the checkpoint data, and runs the corresponding subtasks on workers.

C. Dynamic Data Reloading

As Harmony runs multiple jobs simultaneously, higher memory pressure is inevitable due to the increased number of concurrent jobs. In a managed runtime, such as Java and C#, using a large amount of memory often causes unwanted garbage collection (GC) overheads.

In order to solve this problem, Harmony dynamically spills and reloads input data to/from disks. Within subtask execution model, we can put down the most of input data to disk, because only a single COMP subtask runs at a time even if there exist multiple co-located jobs. Though data reloading can save much memory resources, we need to meet the following requirements not to hinder performance. First, data should be preloaded so as to not block task progress. Second, the total amount of data to reload should be minimized, since it requires additional overheads (e.g., deserialization). To resolve the problem, we designate a portion of data to be in disk and perform spill/reload only for disk-side data, instead of all data. While processing data in memory, we can reload disk-side data in background.

To facilitate the overall management of data, Harmony manages data as fine-grained blocks in memory and on disks. We express the ratio of job j as $\alpha_j = \frac{B_{disk_j}}{B_{total_j}}$, where B_{disk_j} and B_{total_j} represent the number of input data blocks of job j on disk and in total, respectively. During runtime, Harmony keeps adjusting the block ratio to find its optimal value. Increasing α_j makes more amount of data to be spilled and reloaded, which brings additional overhead (e.g., deserialization). We aim to use as least number of disk-blocks B_{disk_j} as possible, while preventing memory pressures and GC overheads. We use hill-climbing method to incrementally move α_j to an optimal value. We determine the initial value by estimating the memory use for accomodating input data and model data. We calculate the size of input and model data by sampling.

V. Evaluation

We have implemented Harmony with 8.8K lines of Java code. We have built Harmony on top of Apache REEF [28] that provides common functionalities for writing distributed systems. In this section we evaluate the performance of Harmony including its scheduling algorithm.

A. Baselines

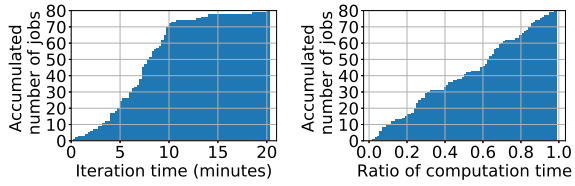
Throughout the experimental results, we provide the following two performance baselines for Harmony:

Isolated: The isolated baseline allocates disjoint sets of resources for each distinct job. In the isolated approach, we try to maximize the CPU utilization rates, as it determines the actual training progress of each job, by reducing the network overheads that occur with lower DoP. Existing works that take similar approaches for allocating resources to each job include Optimus [4] and SLAQ [5].

Naively co-located: The naively co-located baseline naively shares resources between the co-located jobs. In this setting,

Apps	Domain	Dataset	Input (in GBs)	Model (in GBs)
Non-negative Matrix Factorization (NMF)	Recommendation	Netflix64x [29]	45.6	1.0
		Netflix128x	91.2	5.0
Latent Dirichlet Allocation (LDA)	Topic modeling	PubMed [30]	4.3	2.1
		NyTimes [30]	0.6	1.1
Multinomial Logistic Regression (MLR)	Classification	Synthetic [31]	78.4, 155.0	12.0, 24.0
Lasso	Regression			

TABLE I: Workloads used for evaluation. In MLR and Lasso, we use a script for generating synthetic datasets included in Bösen.



(a) Iteration time. **(b)** Comp time / iteration time.

Fig. 9: Key characteristics of workload used for evaluation. We use DoP 16 for all experiments in this figure.

the different combinations of jobs and the different allocations of resources cause greater variance in the performance compared to the isolated baseline. We run all possible cases, and report the best and the worst case. This baseline represents the approach introduced in Gandiva [16], which has no fine coordination between co-located jobs and an analytical basis for job grouping. In our baseline, the minor optimizations in Gandiva for finding better match of jobs are neglected, as we present the best choice obtained from the exhaustive search.

As the baseline systems mentioned above are not open-sourced at the time of submission of the paper, we implement their scheduling schemes on Harmony.

B. Experimental Setup

We run experiments on 100 m4.2xlarge EC2 instances, each with 8 vCPU cores, 32 GB memory and 1.1 Gbps network. On each instance, we co-locate a server and a worker, and one extra instance is used as the master.

As specified in Table I, we use 4 applications each with 2 datasets and 10 different hyper-parameters, resulting the 80 different (app, dataset, hyper-params) tuples. Figure 9 illustrates the distribution of workload characteristics such as the iteration time and the computation to communication ratio.

We run each job until the model convergence. We monitor the objective value (e.g., log-likelihood for LDA, and L2-loss for NMF/MLR/Lasso) at the end of every epoch and determine the convergence by comparing the objective value with the pre-defined threshold. The average CPU and network utilization are measured with an 1-minute interval. For memory resources, we report the GC time during execution, which represents to which extent Harmony relieves the increased memory pressure caused by co-located jobs.

At the baseline of a single job execution in isolation, we confirm that the PS implementation and the machine learning algorithms used in Harmony show similar performances with Bösen [31], an open-source PS system, with its staleness parameter set to 0 for synchronous training. With this condition

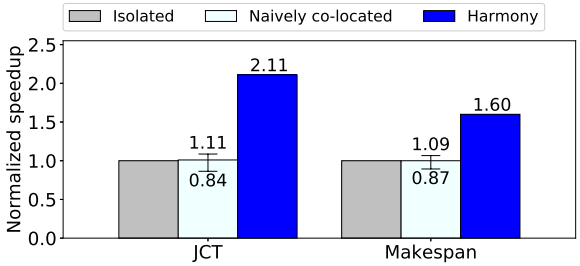


Fig. 10: JCT and makespan in Harmony and the baseline approaches. For naively co-located approach, the bar means average value and the error bar represents max/min values.

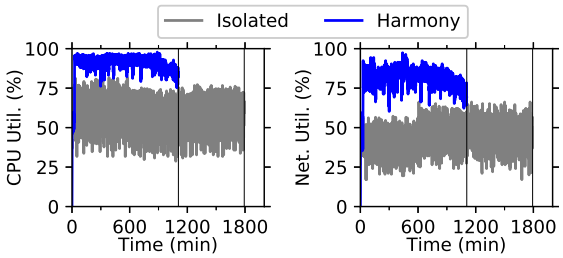


Fig. 11: Resource utilization of Harmony and isolated-approach during an experiment that runs 80 jobs. The vertical lines represent the completion time for all jobs (i.e., makespan).

set, we compare the performances of the scheduling methods in our main evaluations.

C. Performance Comparison

In this section, we compare the performance of Harmony with the other baseline approaches in terms of makespan and average job completion times (JCTs). Concretely, makespan is the time to complete all 80 jobs from the start of the first job, whereas JCT of a job is the elapsed time between the submission and the termination of the particular job.

We show the results in Figure 10, where the makespan and JCT are normalized by the baseline *isolated* approach. First, *naively co-located* approach is 11% and 9% faster in JCT and makespan, respectively, due to the reduced idle time from the co-location of jobs. However, the improvement is limited, as jobs contend with each other for the resources. In the worst case, it is even slower than the isolated approach.

Lastly, Harmony achieves a 2.11× speedup in terms of JCT and 1.60× in makespan with higher utilization of resources, where the regrouping overhead is below 2% of the overall makespan. Figure 11 shows that Harmony shows higher resource utilization and less fluctuating utilization patterns. Harmony achieves average utilization of 93.2% CPU and 83.1% network resources, which is 1.65× higher than the isolated approach. Note that our scheduling can achieve higher network utilizations with further optimizations in the communication layer (e.g., minimizing the serialization overhead). Both the CPU and network utilizations decrease near the end of the execution with smaller number of jobs to co-locate, after the termination of earlier jobs. Note that during an entire execution, 27.2 concurrent jobs were `running` together on

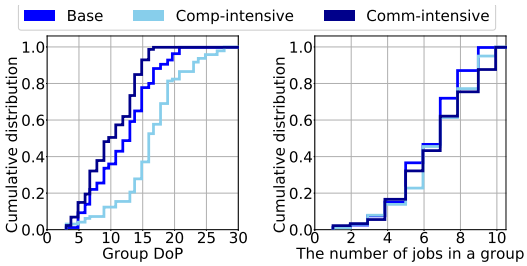


Fig. 12: Distribution of group DoPs and the number of jobs in a group. We extract the information from grouping decisions of the scheduler during whole execution.

average, while divided into 6.7 job groups across all 100 machines.

We investigate how the individual techniques of Harmony contribute to the overall performance benefit. We compare the performance by gradually adding the different techniques on top of each other. With only subtasks (§IV-A), we achieve 32% of total benefit, and adding grouping techniques (§IV-B) achieves 81%, and adding dynamic reloading technique (§IV-C) completes our solution.

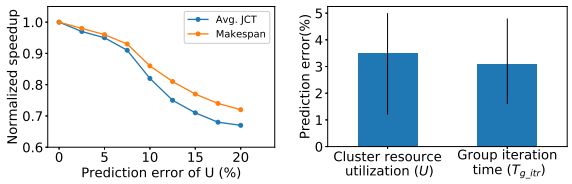
D. Workload Sensitivity Analysis

To show that Harmony can work well with diverse workload, we run two experiments with varying resource usage ratios of jobs and job arrival rates, respectively.

Workloads with different resource usage ratios: We use two different sets chosen from the base workload with 80 jobs. The top and bottom 60 jobs are chosen based on the ratio of computation to communication time (Figure 9b). As a result, the two set of jobs are relatively computation-heavy and communication-heavy compared to the base workload.

The computation-intensive workload runs faster with 1.58× improvement in makespan with 90.5% CPU and 82.1% network utilization in average. The communication-intensive workload also shows 1.57× makespan speedup with 91.8% CPU and 80.9% network utilization in average. From the results, we can see that Harmony successfully achieves high resource utilization regardless of the workload characteristics. It is because Harmony can dynamically determine the average DoP and the entry of running jobs to balance out the computation and communication of running jobs in each group.

The difference comes from the improvement of the average JCT. The computation-intensive workload shows 2.31× speedup of average JCT, but the communication-intensive workload shows 1.83× speedup. We found that the reason is that Harmony uses different DoPs and the number of concurrently running jobs depending on the characteristics. Harmony uses larger DoPs for the computation-intensive workload and smaller DoPs for communication-intensive workload as illustrated in the left graph of Figure 12. Larger DoPs mean smaller number of job groups and subsequently the smaller number of concurrently running jobs. The number of jobs in a group stay rather indifferent to the varying workload characteristics as illustrated in the right graph of Figure 12.



(a) Error sensitivity. **(b)** Prediction error.

Fig. 13: Accuracy of performance model. The vertical line represents the min/max values.

Workload with different job arrival rates: In this experiment, we vary the arrival rate of the base workload for the same set of jobs. We submit jobs with arrival times that follow a Poisson distribution, increasing the mean job arrival time from 0 to 8 minutes. 0 arrival time means that we sumit all jobs at once as the main experiment in §V-C, which shows 2.11× and 1.60× speedup in terms of JCT and makespan, respectively. When we increase the mean job arrival time, the performance starts to decrease slightly from 4 minutes due to the decreasing number of concurrent jobs, and shows 2.01× speedup of avg. JCT and 1.56× speedup of makespan at 8 minutes.

Lastly, we use job arrival rates processes from Google cluster workload traces [32]. We extract 10 job arrival processes randomly from different time windows. While the traces have more diverse pattern of arrivals and job arrival spikes, Harmony handles them well, showing 2.02× speedup of avg. JCT and 1.57× speedup of makespan in average.

E. Accuracy of the Performance Model

To show how important the accuracy of the performance model is, we simulate the execution with different error levels. Figure 13a shows that Harmony manages to provide over 90% of speedup with relatively small errors under 7.5%. However, the performance of Harmony rapidly degrades with larger error levels. It means that the high accuracy of the performance model is crucial for multi-job performance.

We evaluate the accuracy of our performance model, which is used by the scheduler to predict the group iteration time and the resource utilization of the multiplexed jobs. We measure the prediction error by comparing the actual performance and the predicted performance for all scheduling decisions made during all experiments in §V. Thanks to subtask execution model, the prediction error stays below 5% at all times as illustrated in Figure 13b.

F. Performance and Scalability of the Scheduling Algorithm

We evaluate Harmony's scheduling algorithm with an exhaustive search that finds the ground truth that maximizes resource utilization by measuring all possible search spaces. We compare (1) how close the Harmony scheduling decision is compared to the ground truth, and (2) how long it takes to accomplish the scheduling algorithm.

Figure 14 shows the comparison result of resource utilization, average JCT, and makespan, between the solution found with the exhaustive search and the one provided by

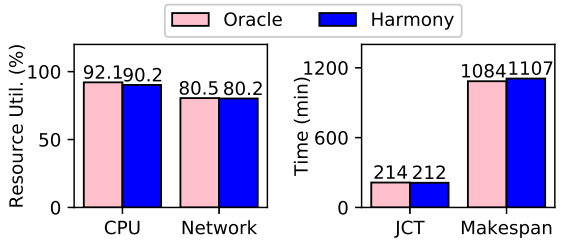


Fig. 14: Comparison of resource utilization, average JCT, and makespan to exhaustive search (Oracle).

Harmony. We see that the results in Harmony is slightly worse by up to around 2%. The difference comes from the fact that our scheduling algorithm finds its solution in a greedy way with a preference of running smaller number of jobs together. This prevents us from exploring the problem space further. However, as shown in the results, the difference is insignificant and this simplification leads to a much higher scalability.

In the experiment above, we run scheduling algorithms for running 80 jobs on 100 machines. The average time to run the scheduling algorithm during overall execution is 1.2 seconds in Harmony and 13.8 minutes in Oracle. To test on a large-scale environment (e.g., datacenters), we emulate the submission and scheduling of thousands of jobs to thousands of machines. According to the result, Harmony can schedule 8K jobs to 10K machines within 5 seconds. This result is comparable to the performance of a scheduler developed recently [4] and a default scheduler of a general RM [33]. On the other hand, the exhaustive search algorithm for 4K jobs on 10K machines takes about 10 hours, due to the exponential growth of the running time of the scheduling algorithm.

G. Dynamic Data Reloading

We perform micro-benchmarks on dynamic data reloading. To evaluate the capability of dynamic adaptation of disk block ratio (α_j) for each job j, we set a baseline that uses the same fixed α for all jobs. In this experiment, we run 8 jobs (4 apps * 2 datasets) on 32 EC2 instances. The result shows that when α is too high, group iteration slows down due to the time of task being blocked by loading corresponding input data blocks. When α is too low, GC explodes and slows down the execution. By running the workload multiple times with different αs, we found the minimum iteration time of 52.9s at $\alpha = 0.3$. Harmony achieves a 44.3s iteration time automatically, which is 16.3% shorter than the manually discovered value in the baseline. The difference comes from the fact that Harmony can dynamically adjust the ratio using different ratios for each job.

In our main experiment in §V-C, the average value of α is 0.34 and has a maximum of 1 and a minimum of 0.11. For only three job groups made by the scheduler has a job with α value 1. To relieve the memory pressure further, Harmony enables spill/reload of model data for those jobs. Though the model data spill/reload is activated just a few times, we confirm that it successfully prevents critical failures (e.g. OOM errors).

VI. Discussion

Fault tolerance: Harmony employs standard failure handling of ML training such as checkpointing (per epoch) and restart. In addition to this, Harmony tries to prevent failures of an individual job from affecting other co-located jobs. For example, the shared runtime catches all exceptions and handles them to prevent the system from crashing. However, a machine/process failure (e.g., OOM) may have an impact on all co-located jobs.

Multi-tenant cluster environment: We assume that we obtain stable performance metrics, which will be used by our performance model. However, in shared cluster environment, the system may show unstable performance occasionally due to interference (e.g., bursty traffics by other users) [34]. In future, our work could be extended to dynamically respond to temporal and permanent changes in profiled metrics.

Other communication architectures: Although Harmony focuses on the PS architecture in this paper, its scheduling approach can be easily applied to other communication architecture such as all-reduce [35], because Harmony does not care how exactly communication is done and only cares that there are distinct computation and communication steps.

Deep-learning (DL) workload: In this paper, we have focused on non-DL workload. Our idea of multiplexing multiple jobs can be extended to DL workload, since the execution pattern of algorithms (i.e., alternating sequence of computation and communication) and commonly used architectures (e.g., PS, All-Reduce) are similar to those of classical ML workload. The biggest challenge is that DL workload typically uses GPU resources, which are not designed to be shared in a fine-grained manner [16].

VII. Related Work

Many recent researches have introduced scheduling solutions specialized to ML workloads [4]–[8]. All of them have greatly improved the cluster performance, but most of them do not consider co-location of multiple jobs into the same resource unit. As a result, Harmony can be used in complementary to the above systems, and vice versa.

Gandiva [16] has been suggested to support co-location of DL jobs into the same GPU. However, Gandiva lacks a clear performance model of co-located jobs, as the interference makes the performance unpredictable. This black-box approach results limited performance gain or loss in some cases. Zhang et al. [36] solves the resource under-utilization problem in datacenters by co-locating batch jobs and latency-sensitive jobs. Unlike Harmony, they handle two different types of workloads that have different schedule priorities.

Zhang et al. [36] and Morpheus [37] use historical information of repetitive jobs. For accurate modeling, Morpheus focuses on mitigating performance unpredictability, like Harmony, but only for periodic workloads using recurring reservations. Harmony, on the other hand, provides analytical performance model and uses online metrics without requiring historical job information, making the system resilient to new ML applications.

VIII. Conclusion

Harmony is a scheduling framework optimized for multiple PS ML jobs to improve cluster resource utilization. Harmony co-locates jobs and coordinates them to share resources effectively by minimizing contention of co-located jobs with subtask execution model. To co-locate jobs that have complementary resource use, Harmony dynamically groups jobs based on the performance model with runtime-collected metrics, adapting to changing pool of jobs. In addition, Harmony alleviates the increased memory pressure with dynamic data reloading. We show that Harmony outperforms existing scheduling approaches and is scalable enough to schedule large-scale workloads. Harmony is open-sourced and publicly available at https://github.com/snuspl/harmony.

Acknowledgement

We thank all reviewers for their comments. This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (2015-0-00221) (2017-0-01772) (2020-0-01649), and by BK21 FOUR Intelligence Computing funded by National Research Foundation of Korea (NRF) (4199990214639), and by Samsung Advanced Institute of Technology (20-0139).

References

[1] Zhipeng Zhang, Jiawei Jiang, Wentao Wu, Ce Zhang, Lele Yu, and Bin Cui. MLlib*: Fast training of GLMs using Spark MLlib. In *ICDE*, pages 1778–1789. IEEE, 2019.

[2] Zhipeng Zhang, Bin Cui, Yingxia Shao, Lele Yu, Jiawei Jiang, and Xupeng Miao. PS2: Parameter Server on Spark. In *SIGMOD*, pages 376–388. ACM, 2019.

[3] Guolin Ke, Zhenhui Xu, Jia Zhang, Jiang Bian, and Tie-Yan Liu. DeepGBM: A Deep Learning Framework Distilled by GBDT for Online Prediction Tasks. In *SIGKDD*, pages 384–394. ACM, 2019.

[4] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *EuroSys*, 2018.

[5] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J Freedman. SLAQ: quality-driven scheduling for distributed machine learning. In *SoCC*. ACM, 2017.

[6] Yixin Bao, Yanghua Peng, Chuan Wu, and Zongpeng Li. Online job scheduling in distributed machine learning clusters. In *INFOCOM*, pages 495–503. IEEE, 2018.

[7] Peng Sun, Yonggang Wen, Nguyen Binh Duong Ta, and Shengen Yan. Towards distributed machine learning in shared clusters: A dynamically-partitioned approach. In *SMARTCOMP*, pages 1–6. IEEE, 2017.

[8] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *NSDI*, pages 485–500. USENIX, 2019.

[9] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*. USENIX, 2014.

[10] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*. USENIX, 2014.

[11] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *SoCC*. ACM, 2015.

[12] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R. Ganger, and Phillip B. Gibbons. Proteus: Agile ML Elasticity Through Tiered Reliability in Dynamic Resource Markets. In *EuroSys*, 2017.

[13] Woo-Yeon Lee, Yunseong Lee, Joo Seong Jeong, Gyeong-In Yu, Joo Yeon Kim, Ho Jin Park, Beomyeol Jeon, Wonwook Song, Gunhee Kim, Markus Weimer, et al. Automating system configuration of distributed machine learning. In *ICDCS*, pages 2057–2067. IEEE, 2019.

[14] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In *NIPS*, pages 19–27, 2014.

[15] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *ATC*, pages 947–960. USENIX, 2019.

[16] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, et al. Gandiva: introspective cluster scheduling for deep learning. In *OSDI*. USENIX, 2018.

[17] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth Gibson, and Eric P Xing. High-performance distributed ML at scale through parameter server consistency models. In *AAAI*, 2015.

[18] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *NIPS*, 2013.

[19] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Exploiting bounded staleness to speed up big data analytics. In *ATC*. USENIX, 2014.

[20] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. In *ICLR Workshop Track*, 2016.

[21] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *EuroSys*, pages 1–16, 2016.

[22] Jason Jinquan Dai, Yiheng Wang, Xin Qiu, Ding Ding, Yao Zhang, Yanzhang Wang, Xianyan Jia, et al. BigDL: A distributed deep learning framework for big data. In *SoCC*, pages 50–60. ACM, 2019.

[23] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R Ganger, Garth Gibson, Kimberly Keeton, and Eric P Xing. Solving the straggler problem with bounded staleness. In *HotOS*. USENIX, 2013.

[24] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. Heterogeneity-aware distributed parameter servers. In *SIGMOD*, pages 463–478. ACM, 2017.

[25] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283. USENIX, 2016.

[26] Suhas Jayaram Subramanya, Harsha Vardhan Simhadri, Srajan Garg, Anil Kag, and Venkatesh Balasubramanian. Blas-on-flash: An efficient alternative for large scale ML training and inference? In *NSDI*. USENIX, 2019.

[27] Ahmed Elgohary, Matthias Boehm, Peter J Haas, Frederick R Reiss, and Berthold Reinwald. Compressed linear algebra for large-scale machine learning. *The VLDB Journal*, 27(5):719–744, 2018.

[28] Byung-Gon Chun, Tyson Condie, Yingda Chen, Brian Cho, Andrew Chung, Carlo Curino, Chris Douglas, et al. Apache REEF: Retainable evaluator execution framework. *ACM TOCS*, 35(2):1–31, 2017.

[29] James Bennett and Stan Lanning. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35, 2007.

[30] Dua Dheeru and Efi Karra Taniskidou. UCI ML repository, 2017.

[31] Carnegie Mellon University. Petuum Bösen, 2016. https://github.com/sailing-pmls/bosen.

[32] John Wilkes and Charles Reiss. Google cluster traces, 2015. https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md.

[33] Tyczynski Wojciech. Kubernetes scalability, 2017. https://kubernetes.io/2017/03/scalability-updates-in-kubernetes-1.6.html.

[34] Jeffrey Dean and Luiz André Barroso. The tail at scale. *CACM*, 2013.

[35] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. Parallax: Sparsity-aware data parallel training of deep neural networks. In *EuroSys*, page 43. ACM, 2019.

[36] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Íñigo Goiri, and Ricardo Bianchini. History-based harvesting of spare cycles and storage in large-scale datacenters. In *OSDI*, pages 755–770. USENIX, 2016.

[37] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, et al. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *OSDI*, pages 117–134. USENIX, 2016.