Ph.D. DISSERTATION

Runtime Optimization Techniques for Resource-Efficient Execution of Distributed Machine Learning

분산 기계 학습의 자원 효율적인 수행을 위한 동적 최적화 기술

FEBRUARY 2020

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Woo-Yeon Lee

Ph.D. DISSERTATION

Runtime Optimization Techniques for Resource-Efficient Execution of Distributed Machine Learning

분산 기계 학습의 자원 효율적인 수행을 위한 동적 최적화 기술

FEBRUARY 2020

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Woo-Yeon Lee

Runtime Optimization Techniques for Resource-Efficient Execution of Distributed Machine Learning

분산 기계 학습의 자원 효율적인 수행을 위한 동적 최적화 기술

지도교수 전병곤

이 논문을 공학박사학위논문으로 제출함

2019 년 11 월

서울대학교 대학원

컴퓨터 공학부

이우연

이우연의 박사학위논문을 인준함

2019 년 12 월

위 원 장	염헌영	(인)
부위원장	전병곤	(인)
위 원	엄현상	(인)
위 원	이재욱	(인)
위 원	전명재	(인)

Abstract

System-level Techniques for Resource-efficient Execution of Distributed Machine Learning

Woo-Yeon Lee
School of Computer Science Engineering
Collage of Engineering
The Graduate School
Seoul National University

Machine Learning (ML) systems are widely used to extract insights from data. Ever increasing dataset sizes and model complexity gave rise to many efforts towards efficient distributed machine learning systems. One of the popular approaches to support large-scale data and complicated models is the parameter server (PS) approach [2, 31, 40]. In this approach, a training job runs with distributed *worker* and *server* tasks, where workers iteratively compute gradients to update the global model parameters that are kept in servers.

To improve the PS system performance, this dissertation proposes two solutions that automatically optimize resource efficiency and system performance. First, we propose a solution that optimizes the resource configuration and workload partitioning of distributed ML training on PS system. To find the best configuration, we build an Optimizer based on a cost model that works with online metrics. To efficiently apply decisions by Optimizer, we design our runtime elastic to perform reconfiguration in the background with minimal overhead.

The second solution optimizes the scheduling of resources and tasks of multiple

ML training jobs in a shared cluster. Specifically, we co-locate jobs with complementary resource use to increase resource utilization, while executing their tasks with fine-grained unit to avoid resource contention. To alleviate memory pressure by co-located jobs, we enable dynamic spill/reload of data, which adaptively changes the ratio of data between disk and memory.

We build a working system that implements our approaches. The above two solutions are implemented in the same system and share the runtime part that can dynamically migrate jobs between machines and reallocate machine resources. We evaluate our system with popular ML applications to verify the effectiveness of our solutions.

Keywords: Machine Learning, Distributed Training, Parameter Server, Cost-based Performance Modeling, Dynamic Optimization, Job/Resource Scheduling
Student Number: 2013-23132

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Distributed Machine Learning on Parameter Servers

Machine learning (ML) training is one of the most popular data processing workloads in datacenters today. Due to the resource-intensive nature, ML workloads typically run on distributed systems that provide more resources, based on the Parameter Server (PS) architecture, which is widely used in both research and industrial communities [3, 10, 11, 28, 36, 40, 42, 45, 46, 54, 62].

In PS-based systems, training data is partitioned across workers, while model parameters – which compose the global model being trained – are partitioned across servers. During training, each of the workers computes model updates using the allocated data and sends the model updates to the corresponding servers. Workers then fetch fresh models from servers in order to work with the latest model parameter values. Servers, meanwhile, apply the model updates received from workers and send the latest model parameter values back to workers as inquired. This process occurs iteratively during the course of the ML training job until the global model converges.

Since ML training is resource-intensive and time-consuming, it is important to optimize the system to efficiently use the given resources and improve the system perfor-

mance. Specifically, we focus on two key problems that we describe in the following sections. First, choosing the right PS system configuration is challenging, however the existing systems require users to configure it manually and also assume system configuration to be static. Second, distributed ML training often suffers from inefficient resource use, due to the fact that ML jobs repeat computation and communication steps, each of which uses only a single type of resource intensively. In the following sections, we elaborate on these problems and introduce our corresponding solutions.

1.2 Automating System Configuration of Distributed Machine Learning

The performance of PS system is crucially dependent on choosing the right system configuration: the number of workers and servers as well as the training data and model partitioning across them. Current PS implementations assume system configuration to be static: the configuration is chosen before training commences and remains unchanged until job termination. However, as we illustrate in Section 3.1, choosing the best system configuration is challenging; optimal system configuration parameters vary widely for different algorithms, hyper-parameters, and environments. Furthermore, the best configuration changes during runtime as the total amount of available resource changes.

We present cost-based optimization that finds a good system configuration for PS-based frameworks. We extend a PS-based ML framework to build Cruise that automatically tunes its system configuration with the optimization technique. Cruise focuses on the system aspects and models performance of workers, as an optimization goal, analytically with the system's runtime statistics, and computes optimal configurations by solving the optimization problem. Cruise applies the new configurations efficiently during runtime by elastically changing allocated resources and migrating data. The reconfiguration allows us to make the best use of given resources and also opportunistically available resources. Our evaluation shows that our cost model is valid and

Cruise finds a good system configuration automatically to optimize the performance. With widely-used machine learning workloads, we demonstrate that the configuration found by Cruise performs close to the optimal configuration that we find exhaustively, with the difference at most 6.5%. Cruise reduces the training time by up to 58.3% compared to static configuration within tens of seconds reconfiguration overhead.

1.3 Scheduling of Multiple Distributed Machine Learning Jobs

Due to the resource-intensive nature, ML workloads are often executed on shared clusters to utilize the given resources more efficiently [3,25,33,43,45,58]. In such settings, efficient resource scheduling among the ML jobs is key to improving cluster-wide performance. The problem is that an ML training job does not fully utilize the allocated resources, as each step of an ML job intensively uses a particular type of resource while leaving the others mostly idle, resulting in an average utilization of around 50% of the overall assigned CPU and network resources [33, 40, 41, 58]. Concretely, ML workloads consist of iterations that each repeat computation and communication steps, using different resource alternately and causing inefficiency in the overall resource utilization.

In order to improve resource utilization by breaking the barrier between the computation and synchronization steps, a number of works have proposed asynchronous training methods with local model cache that disintegrate the sequential dependency of the different steps [15, 19, 31]. In such works, workers pull recent 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 of stale models, and hence hinders model convergence [9,16,18,19]. Although many works try to minimize stale models in asynchronous training by constraining maximum staleness [14, 15, 31] or by differentiating the learning rate of the delayed updates [34], they have been able to reduce the side-effects but not completely resolve the issue, occasionally showing

worse performance for complex models [16]. Due to such reasons, many works retain synchronous training to avoid staleness [1, 9, 16, 18].

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 the same type of resource, if both jobs happen to have to use the same type of resources simultaneously. This can result in an even slower job completion time than running each job alone (§4.4). In addition, while ML training is memory-intensive [23, 49], co-locating multiple jobs incur 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.

For example, a recent system, Gandiva [58] co-locates jobs by simply performing a series of trials-and-errors, which naively avoids contention when it faces one, instead of fundamentally analyzing and eliminating contentions. Gandiva has used several auxiliary techniques to minimize the interference between jobs, but its improvement in resource utilization is limited to only about 10%. Although Gandiva is specialized for scheduling DL jobs running on GPUs, which we leave for future work, interference of co-located jobs play a critical role in non-DL classical ML applications as well, which we focus on in the dissertation. Moreover, Gandiva simply focuses on sharing computation resources (e.g., GPU), and ignores network resources, which play an important role during the parallelization and the model synchronization (communication) step of training jobs.

To overcome these limitations, we introduce Harmony, a scheduling framework that co-locates multiple ML jobs and optimizes resource utilization among them, reducing the average 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 vary 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 is designed to dynamically reschedule jobs and resources to continuously find more efficient groupings and resource allocation. To minimize overhead of continuous regroupings, we design our scheduling algorithm to minimize job movements and 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 pattern. 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 memory pressure and disk read overhead towards minimal training time.

Our evaluation on 64 m4.2xlarge AWS EC2 instances shows that Harmony improves cluster resource utilization by up to 1.65$\times$ 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%.

1.4 Contributions

The proposed approaches and implementations in this dissertation adds new system-level supports for PS-based distributed ML frameworks with the following contributions:

- We identify the performance problem of PS ML systems and resolve the problem by exploiting the execution characteristics of distributed ML training, which has iterative pattern with separate computation and communication steps.
- We build an analytical performance model to predict system performance with runtime-collected metrics, based on the analysis of resource usage pattern in the execution of distributed ML training on PS-based systems.
- We present the practical solution and the complete open-sourced implementation that automatically optimizes the system configuration and job/resource scheduling for the optimized execution of distributed ML training on PS systems. Our solution is general enough to be applied to other PS system implementations.

1.5 Dissertation Structure

This dissertation consists of five chapters. The first chapter presents an introduction and the remainder chapters describe our work in more detail with conclusions and future work.

In Chapter 2, we describe PS ML frameworks, which is a common background for the overall work. Chapter 3 describes our first study for automating system configurations of distributed machine learning frameworks. We explain how we find the optimal configuration and apply it during runtime with minimal overhead. Chapter 4 presents our solution for efficient scheduling of multiple distributed machine learning jobs. We describe how to group jobs with complementary resource use, execute co-located job tasks harmoniously without contention, and alleviate memory pressure due to co-located jobs. Chapter 5 concludes this dissertation with a summary and describes future work.

Chapter 2

Background

ML training is an iterative process that incrementally improves a model until it reaches a certain convergence threshold. To facilitate large-scale ML training in distributed environments, systems designed with the parameter server (PS) architecture have been introduced [11,20,40,59]. As illustrated in Figure 2.1, the PS architecture mainly consists of *server*s that each maintains a partition of ML model parameters, and *worker*s

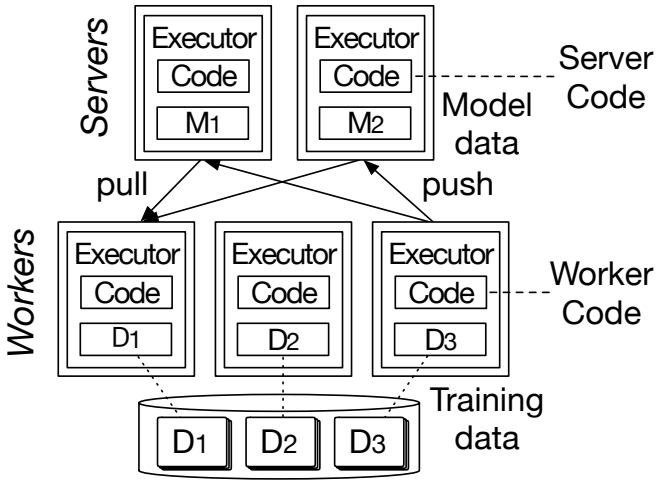


Figure 2.1: Parameter server ML framework

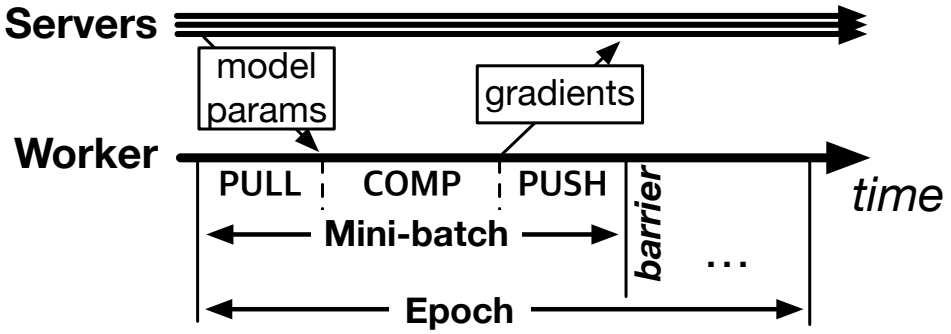


Figure 2.2: The work-flow of a PS system.

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. Therefore, servers mostly utilize network resources to sync ML models with the workers, whereas workers mainly use CPU resources for their computations. To highly utilize both CPU and network resources and to reduce the network overheads, workers and servers are usually located together [28, 54].

In the figure, *Executor* is an environment on which the ML application code (e.g., worker computation code and server model update code) runs. Executor takes care of low-level system supports such as initializing and maintaining network connections between nodes. In this dissertation, we consider a cluster environment where each Executor runs in a container obtained from a Resource Manager such as YARN and Mesos.

Figure 2.2 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*, 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. As ML training repeats same procedure composed of computation and communica-

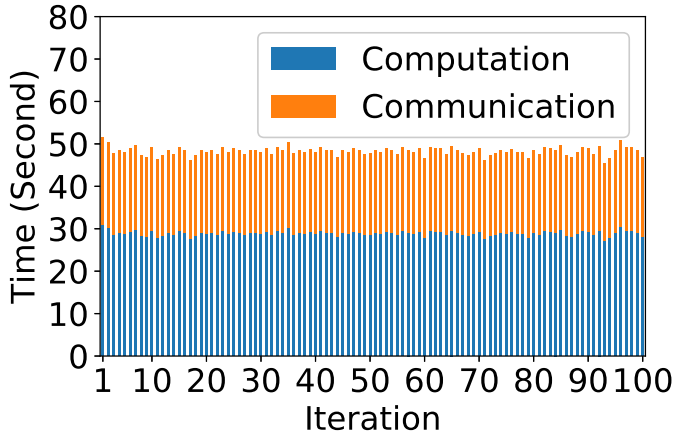


Figure 2.3: Iteration time during 100 training iterations of a NMF job.

Application	NMF	LDA	MLR	Lasso
Iteration time	48.3±1.0 s	94.5±1.8 s	32.4±0.7 s	28.9±0.5 s

Table 2.1: The average and standard deviation of iteration times of ML apps.

tion until the model converges, each iteration takes similar time with computation and communication times as illustrated in Figure 2.3 and Table 2.1.

Chapter 3

Automating System Configuration of Distributed Machine Learning

The performance of distributed machine learning systems is dependent on their system configuration. However, configuring the system for optimal performance is challenging and time consuming even for experts due to the diverse runtime factors such as workloads or the system environment. We present cost-based optimization to automatically find a good system configuration for parameter server (PS) machine learning (ML) frameworks. We design and implement Cruise that applies the optimization technique to tune distributed PS ML execution automatically. Evaluation results on three ML applications verify that Cruise automates the system configuration of the applications to achieve good performance with minor reconfiguration costs.

We organize the rest of this chapter in the following way. In § 3.1, we show the configuration challenges in the parameter server. § 3.2 describes our approach to find the optimal configuration based on cost model with runtime metrics. We then introduce Cruise, which implements our approach and briefly explain the roles of each component in § 3.3. We evaluate Cruise in § 3.4 to demonstrate the performance enhancements achieved by deploying the system, verifying its contributions towards the current state-of-the-art. § 3.5 acknowledges our related work. We summarize our work

App.	NMF	MLR	LDA
(18,14)	1.30x	2.29x	**Best**
(23,9)	**Best**	1.12x	1.21x
(27,5)	1.66x	**Best**	1.60x

(a) Case 1. Epoch time comparison in different algorithms: NMF, MLR, and LDA.

#Topics	400	4K
(18,14)	**Best**	1.44x
(23,9)	1.21x	**Best**
(27,5)	1.60x	1.20x

(b) Case 2. Epoch time comparison in different hyper-parameters in LDA.

Env.	m4. large	m4. xlarge
(34,18)	**Best**	1.10x
(38,14)	1.08x	1.05x
(42,10)	1.48x	**Best**

(c) Case 3. Epoch time comparison in different VM instance types in NMF.

Table 3.1: Epoch times varying numbers of workers and servers for different algorithms, hyper-parameters, and virtual machine instance types.

in § 3.6, with an endeavor to advance the system even further by suggesting potential steps for the near future.

3.1 System Configuration Challenges

The system configuration of a PS system includes the allocation of worker and server roles to available containers, as well as the partitioning of the training data across workers and the model parameters across servers. A good system configuration is essential for the performance of the machine learning system [60]. However, optimal system configurations that produce minimal training time are difficult to find, even for system experts. This is because, first, predicting how ML application code translates to actual running time - how much time each step takes - is nontrivial. Even if we were to estimate the exact running time for an algorithm, there may exist many different implementations for that particular algorithm, all having slightly different running times. Second, even for the same algorithm, using different hyper-parameters can change the application's computation or communication overhead. Finally, the capabilities of the environment on which the applications run vary from cluster to cluster.

We illustrate the challenges with experiments that vary algorithms, hyper-parameters, and machines in Table 3.1. In the table, we use notation (W, S), where W denotes the

number of workers and S denotes the number of servers. For each column, a cell presents a ratio between the epoch time of the configuration and the optimal epoch time. In each case, we fix the total number of machines, assign a fraction of the machines to run workers, and assign the rest of the machines to run servers. We experiment with all possible worker and server configurations to compare epoch time of different configurations.

All experiments in Tables 3.1a and 3.1b were run on a cluster of 32 AWS EC2 r4.xlarge instances (4 CPU vCores, 30.5GB memory, and 1.25 Gbps network bandwidth), and Table 3.1c shows epoch time of an ML application on either a cluster of 52 m4.large instances (2 CPU vCores, 8GB memory, and 0.5 Gbps network bandwidth) or a cluster of 52 m4.xlarge instances (4 CPU vCores, 16GB memory, and 1.0 Gbps network bandwidth). In the table, NMF denotes Non-negative Matrix Factorization, MLR denotes Multinomial Logistic Regression, and LDA denotes Latent Dirichlet Allocation. We present the details of these algorithms in § 3.4.1.

Case 1: ML algorithm. Different ML algorithms show different optimal configurations. From Table 3.1a, with (W:27, S:5) MLR achieves the smallest epoch time, whereas LDA runs 1.6 times slower with this configuration compared to LDA's optimal configuration (W:18, S:14). This is because MLR is more compute-intensive than LDA, thus requiring more workers for smaller epoch time.

Case 2: Hyper-parameter. Hyper-parameter values affect the optimal configuration of an ML application. A hyper-parameter in LDA is the number of topics to categorize documents. Increasing the number of topics makes both computation and communication more expensive, but they are affected differently. Table 3.1b shows that (W:18, S:14) is the best configuration for 400 topics but is 1.44 times slower than the best configuration (W:23, S:9) for 4K topics.

Case 3: Machine environment. The specification of the cluster on which jobs run also heavily affects the best configuration due to varying computation and communication capabilities. Running NMF on different clusters, we observe that the best configuration varies drastically as shown in Table 3.1c. When we use AWS EC2 m4.xlarge

instances, the best configuration is (W:42, S:10), which is 1.48 times slower if the same configuration of m4.large instances, compared to the m4.large best configuration (W:34, S:18).

The three factors investigated above demonstrate that discovering the optimal system configuration is challenging. Even worse, there are other factors such as algorithm implementation and dataset that can also affect the performance for different configurations. Since the problem space is too broad, it is hard to predict the performance of an ML application given a specific setting. This motivates adapting to optimal configurations automatically.

3.2 Finding Good System Configuration

In this section, we describe our cost formulation of the training epoch time along with our model assumptions, and how we minimize epoch time by using the cost model to find values for system configuration parameters – namely, the number of workers and servers as well as the training data and model partitioning across them.

3.2.1 Cost Model

Given the PS architecture, we define the cost C of the entire system to be the maximum of the time for each worker i to process the assigned training data in each epoch (C^i: *epoch time*). By minimizing the maximum epoch time (C), we can improve the absolute performance as well as balancing all workers' performance. Unbalanced training can slow down the learning process because training data does not contribute evenly to the global model parameters. In a general system consisting of heterogeneous containers[1] and uneven data partitions, C^i is usually different for each worker.

$$C = \max_i C^i \tag{3.1}$$

[1]We focus on modeling heterogeneous containers because of heterogeneous hardware or virtual machines. Transient stragglers are not part of the model. We borrow work stealing techniques from prior work [27] to handle stragglers.

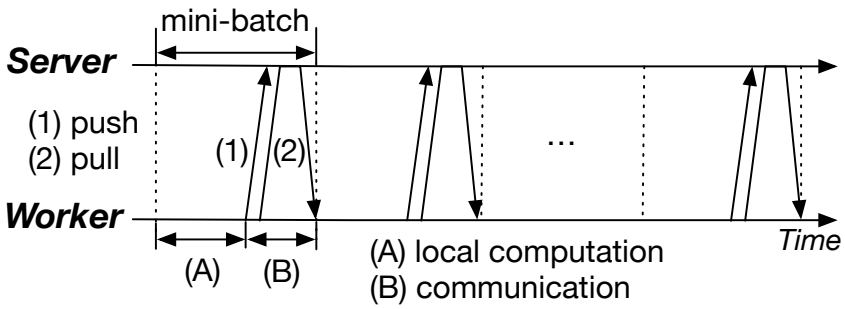


Figure 3.1: A worker's epoch

A worker's epoch can be further split into smaller components. Figure 3.1 depicts the timeline of a worker's epoch. A worker first performs computation on its training data using the current model to produce model gradients. The worker then communicates with the servers to send its gradients via push requests and fetches the updated model via pull requests. Depending on the algorithm and additional job parameters, workers may divide the training data into several smaller subsets and go through a computation-communication cycle for each subset. Such computation-communication cycles are called *mini-batches*. The next epoch begins once the worker has processed all of its mini-batches (i.e., all training data assigned to the worker).

To simplify the cost model, we make the following assumption on communication between workers and servers. First, push requests from workers do not block gradient computation and thus can be sent asynchronously with respect to the workers' local computation. On the other hand, a fresh pull of the whole model must always occur before local computation takes place. We assume such a model where pull requests are issued synchronously and blocks local computation [60]. The synchrony of pull requests can be partially resolved by decoupling the computation mechanism from communication threads [53]; this leads to a different cost formulation that can be understood as a variation of the one described in this section.

We define the total time spent on local computation of an epoch as *computation cost*, and the time spent on the communication as *communication cost* (denoted by (A) and (B) in Figure 3.1, respectively). Communication cost, to be more specific, is the sum of the elapsed times between a push request's initiation and the response

for a successive pull request in each mini-batch. Using C^i_{comp} and C^i_{comm} to denote the computation and communication cost of worker i respectively, the epoch time of worker i becomes

$$C^i = C^i_{comp} + C^i_{comm} \tag{3.2}$$

3.2.2 Cost Formulation

Computation cost. This cost depends on the size of the training dataset and the computing power of workers. The entire dataset of size D is split and distributed to w workers. C^i_{comp} depends on the size of the training dataset d_i assigned to worker i and the computing power of the worker. Depending on the time complexity f of the ML algorithm, C^i_{comp} depends on $f(d_i)$ since a worker-side computation scans all of the allocated training data during an epoch. In case an ML algorithm has linear time complexity (e.g., NMF, MLR, LDA), C^i_{comp} is proportional to d_i. In a general system consisting of heterogeneous containers (e.g., containers with different numbers of cores), each worker i takes $C^i_{w.proc}$, the time spent to perform computation on a single training data instance, which varies across workers.

$$C^i_{comp}(d_i) = C^i_{w.proc} d_i \tag{3.3}$$

$C^i_{w.proc}$ depends on factors such as the implementation of ML algorithm or the hardware of the worker container. In Cruise, $C^i_{w.proc}$ is measured by monitoring workers' local computation; we measure the elapsed time for workers to compute gradients and divide it by the amount of the training data instances. A larger dataset makes the computation cost more expensive, but we can reduce the cost by introducing more workers, which reduces the size of dataset d_i that each worker processes in each epoch.

Communication cost. We model communication cost as the time a worker takes when communicating with the server. The entire model of size M is split and dis-

tributed over s servers. m_j is the size of partial models assigned to server j. We consider the following two cases to model C^i_{comm}.

1. *Server network bandwidth is the bottleneck.* The serving latency of server j is the number of bytes sent to j divided by the bandwidth b_{ij} between worker i and server j: $\frac{m_j w}{b_{ij}}$ where w is the number of workers. With a mini-batch size of B, each worker i executes $\lceil \frac{d_i}{B} \rceil$ mini-batches per epoch. Since the communication cost is determined by the slowest server, $C^i_{comm} = \lceil \frac{d_i}{B} \rceil \max_j (\frac{m_j w}{b_{ij}})$.

2. *Worker network bandwidth is the bottleneck.* In this case, the worker's network bandwidth is being fully utilized to serve push and pull requests. The cost is formed as the number of bytes sent by worker i divided by its bandwidth: $C^i_{comm} = \lceil \frac{d_i}{B} \rceil \sum_j \frac{m_j}{b_{ij}}$.

Then, the communication cost C^i_{comm} is the maximum of the above two terms.

$$C^i_{comm} = \left\lceil \frac{d_i}{B} \right\rceil \max \left(\max_j (\frac{m_j w}{b_{ij}}), \sum_j \frac{m_j}{b_{ij}} \right) \tag{3.4}$$

3.2.3 Optimization

Optimization problem. In Figure 3.2, we formally define our optimization problem. For heterogeneous environments where some machines have higher computing power or network bandwidth. In these environments, the configuration space becomes larger, because we also need to determine the data distribution as well as deciding whether to run a worker or a server on a container.

The optimization goal is to find the parameters $\mathbf{w}, \mathbf{s}, \mathbf{d}, \mathbf{m}$ that minimize the cost function, where $\mathbf{w}$ and $\mathbf{s}$ denote the assignment of machines to workers and servers, respectively, and $\mathbf{d}$ and $\mathbf{m}$ denote the partitioning of the training dataset and partial models, respectively.

Given N machines, we adjust the configurations to meet the optimal balance between computation and communication costs. For example, using more machines as

Given parameters

N: the total number of machines

D: the entire dataset size

M: the entire model size

B: the mini-batch size

Variables

$\mathbf{w} = \{0,1\}^N$: $w_i = 1$ if a worker runs on machine i

$\mathbf{s} = \{0,1\}^N$: $s_j = 1$ if a server runs on machine j

$\mathbf{d} = (d_1, ..., d_N)$: training data partitioning for workers

$\mathbf{m} = (m_1, ..., m_N)$: model partitioning for servers

Problem

Find $\mathbf{w}^*, \mathbf{s}^*, \mathbf{d}^*, \mathbf{m}^*$

$$= \underset{\mathbf{w},\mathbf{s},\mathbf{d},\mathbf{m}}{\operatorname{argmin}} \max_i C^i(\mathbf{w}, \mathbf{s}, \mathbf{d}, \mathbf{m})$$

$$= \underset{\mathbf{w},\mathbf{s},\mathbf{d},\mathbf{m}}{\operatorname{argmin}} \Big[\max_i \Big[C^i_{w.proc} d_i + \lceil \tfrac{d_i}{B} \rceil \max\big(\max_j (\tfrac{m_j \|\mathbf{w}\|}{b_{ij}}), \sum_j \tfrac{m_j}{b_{ij}} \big) \Big] \Big]$$

Constraints

$\|\mathbf{w}\| + \|\mathbf{s}\| = N$: workers and servers are assigned to N machines disjointly

$\sum_i d_i = D$: total number of training data samples

$\sum_j m_j = M$: total number of model partitions

Figure 3.2: Optimization Problem

workers certainly brings down the computation cost by reducing the training data size that each worker deals with. However, this leads to high communication cost due to an increased number of push and pull requests within an epoch and fewer containers available for servers.

Solution. Based on the problem definition in Figure 3.2, we cast the optimization problem as Mixed Integer Programming (MIP) and solves the problem using a solver library from Gurobi [26]. Since the quadratic terms affect the performance significantly, we encode integer variables d and m in binary representation, which allows the solver to multiply variables faster. As a result, the MIP program consists of $O(N^2)$ variables, $O(N)$ quadratic constraints, and objective terms. In case that we have homogeneous machines (i.e., all machines have the same computing power and network

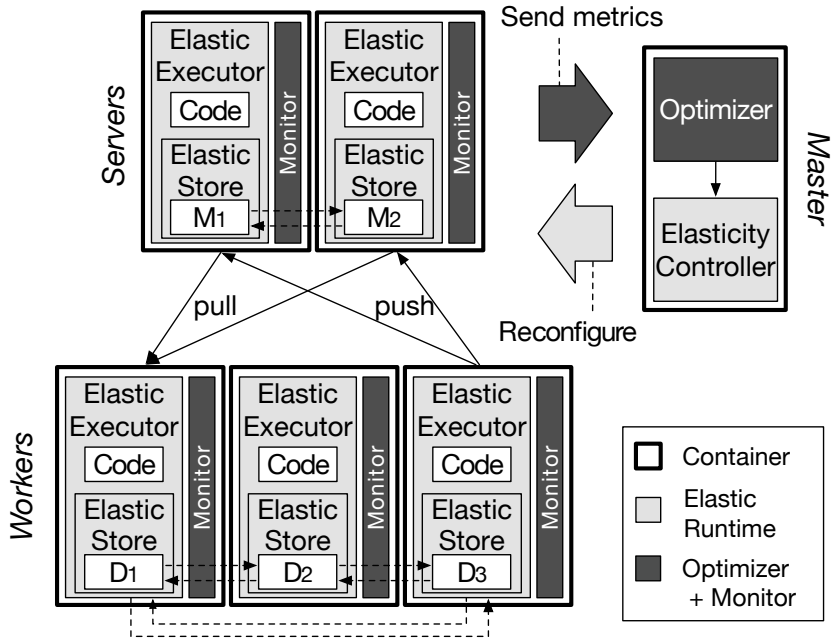


Figure 3.3: Cruise Architecture.

bandwidth), the optimal solution distributes d evenly across workers and m evenly across servers. Thus, we can derive an analytical solution that runs in $O(N)$. We present our analytical solution below, but due to space constraints, we omit its derivation.

$$\begin{aligned}
&\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \Big[\frac{D}{\|\mathbf{w}\|} T(\|\mathbf{w}\|)\Big], \\
&\quad \text{where } T(\|\mathbf{w}\|) = C_{w.proc} + \frac{M}{b} \max(1, \frac{\|\mathbf{w}\|}{N - \|\mathbf{w}\|})/B. \\
&\quad \mathbf{s}^* : s_i = 1 - {\mathbf{w}_i}^*, \quad \mathbf{d}^* : d_i = \frac{D}{\|\mathbf{w}^*\|}, \\
&\quad \mathbf{m}^* : m_j = \frac{M}{N - \|\mathbf{w}^*\|}, \quad b : \text{machines' bandwidth}
\end{aligned} \tag{3.5}$$

3.3 Cruise

We extend an existing PS system to automatically configure distributed ML execution. The extended system called Cruise adds Optimizer and Elastic Runtime to the PS system, as depicted in Figure 4.5. Optimizer estimates the optimal configuration for a running ML job using runtime metrics. Following the decision of Optimizer, Elastic Runtime applies the necessary changes dynamically to the system without stopping

the running job.

3.3.1 Optimizer

Optimizer performs cost-based optimization by solving an optimal configuration problem formulated in § 3.2. Monitors collect runtime statistics related to the performance (e.g., the elapsed time for workers to compute gradients) and reports the metrics to Master periodically. Optimizer then estimates the performance in different system configurations based on the runtime status. By doing so, our optimizer does not require knowledge about the ML jobs (e.g., algorithms and hyper-parameters). After finding the configuration that is expected to be optimal, Optimizer maps the difference from the current configuration and generates an optimization plan, consisting of operations provided by Elastic Runtime. By executing the operations in the plan, Cruise changes the system configuration to the one with better performance. To achieve performance benefit with optimization, we need to make decisions such as when to calculate an optimization plan, whether to execute the plan or not. We describe these policies below.

Metric Collection

Cruise collects runtime metrics to use them as inputs to the Optimizer. Workers measure local computation time and communication time and report to Master at the end of every mini-batch. On the other hand, Servers report the metrics to Master periodically. Since the runtime metrics can fluctuate, we apply moving average to reduce noise.

Optimization Trigger Policy

Based on the cost model above, Cruise triggers optimization after collecting sufficient metrics to substitute the unknown variables in the cost model. We use metrics at the mini-batch granularity to be responsive to the changes of the running job. Using metrics from a configured number of subsequent mini-batches, we estimate the cost of an epoch.

In order to avoid the system from continuously reconfiguring back and forth around the estimated optimum, Optimizer predicts the performance benefit of a new config-

uration and skips that attempt if the gain is less than a certain threshold. A threshold number from our experience -5%- is good enough to prevent the system from "oscillating", while allowing the system to undergo moderately-sized optimizations.

When the amount of available resources (e.g., N) increases, Optimizer opportunistically tries to use the extra resources. If more resources become available, Optimizer can adjust to find an optimal configuration including the new resources. When the amount of available resources decreases, it rebalances execution accordingly.

Optimization Execution

Once the decision of a reconfiguration is made with the computed system configuration $(\mathbf{w}^*, \mathbf{s}^*, \mathbf{d}^*, \mathbf{m}^*)$, the new configuration is contrasted with the current configuration $(\mathbf{w}, \mathbf{s}, \mathbf{d}, \mathbf{m})$ to generate a reconfiguration plan. All plans consist of a subset of four Elastic Runtime operations, which we discuss in detail in § 3.3.2. The operation `add` is for newly joining containers, while the operation `delete` deletes containers that are no longer assigned any data or partial model. The operation `switch` is to change an existing server container to worker or from worker to server. Training data and model partitioning, $(\mathbf{d}, \mathbf{m})$, can be modified by migrating data between containers to preserve the state of the running job, which we will further discuss in § 3.3.2. The `move` operation migrates data between containers, starting with containers that have the largest training data or model changes in a greedy fashion, to minimize the amount of data to move and the number of movements.

Optimizer executes a plan by simply invoking Elastic Runtime API that reconfigures system transparently without stopping training. The simplest approach to execute the plan would be to invoke the operations sequentially. However, to make the reconfiguration agile, Optimizer generates the plan as directed acyclic graphs of independent operations that can be executed concurrently.

3.3.2 Elastic Runtime

Elastic Runtime is an execution environment that exposes operations which Optimizer can call to dynamically reconfigure the system. Elastic Runtime manages workers and servers in the form of containers, each integrated with an *Elastic Executor*. Elastic Executor runs application code on data encapsulated by *Elastic Store*, a distributed key-value store that constructs an effective management scheme. *Elasticity Controller* manages the distributed Elastic Executors. It is also the endpoint where Optimizer triggers reconfigurations according to the generated optimization plan.

Elastic Runtime deals with two types of reconfigurations: resource reconfiguration and workload repartitioning. Resource reconfiguration is achieved by Elastic Executor, a containerized and reconfigurable runtime which extends the existing PS architecture's Executor. Elasticity Controller coordinates resource reconfiguration by easily adding and removing Elastic Executors. Workload repartitioning is conducted efficiently with Elastic Executor's internal component, Elastic Store. An Elastic Store encapsulates data in an in-memory storage with a management scheme that provides flexibility in the accommodated data type (e.g., training data or model data).

Transparency must be maintained in the course of a reconfiguration. Reconfiguration must occur with minimal effects to the running job by maintaining the application's access to data without any loss or significant overhead. Elastic Executor performs several additional tasks required for a transparent reconfiguration, such as adaptive data ownership management or redirection of requests to the new owner of data.

We explain the details on resource reconfiguration in § 3.3.2 and workload repartitioning in § 3.3.2 while maintaining transparency in § 3.3.2.

Resource Reconfiguration

Containers can be added or deleted when Optimizer determines so with operations `add` and `delete` for which the simple signatures are provided in Figure 3.2. When `add` is executed, Elastic Runtime simply launches an Elastic Executor on a new container.

Data access interface	Description
`put(Key, Value)`	Puts (`Key`,`Value`) to Elastic Store
`get(Key)`	Gets the value associated with `Key`
`update(Key, Func, Delta)`	Updates the value for `Key` with the result of `Func(Value, Delta)`
Reconfiguration interface	**Description**
`add(ResourceConf, RuntimeConf)`	Adds new containers and starts runtime on them
`delete(Containers)`	Deletes existing containers
`switch(Container, RuntimeConf)`	Switches a container to run a specified runtime
`move(Blocks, SrcContainer, DstContainer)`	Moves blocks from one container to another

Table 3.2: Elastic Runtime Interfaces.

In the case of a container `delete`, Elastic Executor stops the app code and itself to release the container.

When deleting an executor, Elastic Runtime performs additional wrap up, corresponding to its role (e.g., worker or server). Before a server-side Elastic Executor shuts down, it redirects all remaining pull requests from workers to the new owning Elastic Executors to prevent workers from waiting long for a response. For worker-side, it waits until ongoing mini-batch to be finished and push requests are flushed to servers.

Elastic Runtime also provides `switch` operation that changes Elastic Executor to another type (e.g., from server to worker or from worker to server). This operation also involves the setup and cleanup procedure involved in `add` and `delete`. However, the two procedures occur in parallel in the existing container and there is no container setup or cleanup involved. This is especially beneficial in an environment with constrained resources as `add` must wait for a container to become free after a `delete` completes.

Workload Repartitioning

Workload repartitioning includes changing each container's ownership of training data/-partial models and migrating the data accordingly. A resource reconfiguration must occur in conjunction with workload repartitioning. When a container is added/deleted, the workload for each container must be readjusted across the new set of containers

now running in the system. Workload repartitioning may occur on its own in the case of an imbalance in workload between containers.

Any runtime state of the job such as the model data across servers must be preserved, not to lose the job's progress. Thus, the states must be migrated from one container to another. In addition to such mutable data, training data across workers which remains unchanged can enjoy the benefit of migration to reduce the overhead of reloading the entire dataset in workload repartitioning. Both mutable and immutable data can be stored in Elastic Stores on which workload repartitioning occurs.

Data Storage and Ownership Management: Data management in Elastic Runtime involves a collection of Elastic Stores where the actual key-value tuples are stored. The actual ownership of each data instance is maintained by the respective Elastic Store, but Elasticity Controller also maintains a global ownership view to orchestrate migration between Elastic Stores. Ownership tables are updated during the migration process, which we will discuss the details below in this section.

Elastic Stores are composed of *blocks* containing data and a block is owned by exactly one Elastic Store. The entire key-space of data is partitioned and each block contains data for a range in the key-space. For an even partitioning of keys over blocks, each block stores data for a hashed key range. Clients of Elastic Stores - worker and server code - use a *key* which is mapped to a value to access each key-value tuple. For each client access, the only Elastic Store owning the block where the key-value tuple is stored processes the request according to the *ownership table*.

Data Access: Elastic Runtime allows values to be stored to and retrieved from Elastic Stores through simple operations, similar to what can be done in distributed hash tables (DHTs) [21]. The difference of Elastic Stores over such key-value stores is that Elastic Runtime exposes options to migrate data. Elastic Store provides simple and standard operations for clients to access and update each data instance with a key as shown in Figure 3.2. Elastic Runtime guarantees that operations are served exactly once by maintaining a single owner of the block containing the key-value tuple on which the operation is conducted across all Elastic Stores. When an operation is

requested to the Elastic Store that does not own the block, the request is processed by remotely accessing the owner according to the ownership table in each Elastic Store.

In addition to the `put/get` operations, we provide `update` operation, which atomically executes `Func`, a user-defined function that should be commutative and accumulative to guarantee atomic incremental updates.

In Cruise, when starting a job, a worker Elastic Executor loads its assigned set of training data using `put` into its local Elastic Store. While running the job, Elastic Executor fetches the data to process for each mini-batch from the local Elastic Store using `get`. Servers, however, must use `update` when processing a push request to guarantee atomicity. To process a pull request, servers simply `get` model data from the local Elastic Store.

Data Migration: `move` operation changes ownership and migrates data between Elastic Stores. This should be done carefully to prevent loss or duplicated processing of an operation, while changing block owner. It is also the most critical factor that determines reconfiguration performance and thus Elastic Runtime executes multiple `moves` concurrently, each `move` parallelized in block units.

We implement the following protocol in Elastic Runtime to provide an efficient migration process. Elasticity Controller initiates a migration for a set of blocks by sending a message to the source container. The source container migrates blocks concurrently to the destination container and reports Elasticity Controller about the completion of the migration for every block, upon each ACK message from the destination container. Finally, Elasticity Controller broadcasts ownership change of the block to all other containers. Specifically, the block migration is done in two distinct steps: ownership handover and actual data transfer. In the source container, when starting migration for a block it hands over ownership first, so access operations for the block in this container are redirected to the destination container. In the destination container, when it takes an ownership it starts queueing access operations for the block and starts processing them after receiving actual block data.

The key point in the migration process is that block ownership is transferred atom-

ically such that there is always a single owner for a block. Another key point is that even if multiple blocks are requested for migration, client access to a key is blocked only during the actual migration of the block containing the key.

Transparency during Reconfiguration

Dynamic reconfiguration must occur without any extra work for Elastic Runtime's clients, but also must refrain from any performance degradation. Such *transparent* reconfigurations include the following requirements. First, client access APIs must be supported during reconfiguration, maintaining read-my-write consistency. Second, in effort to serve client access, overheads such as increased number of remote data accesses are inevitable due to resource reconfigurations. Such inefficiency must be minimized. Finally, the reconfiguration must guarantee that the accuracy of the model being learned is unaffected. Elastic Runtime reinforces the key requirements in achieving transparent reconfigurations with the following features.

Data accessibility: Data must be accessible any time during and after data migration for client access. Elastic Store enables remote access with the ownership table maintained atomically during the migration process.

Data locality: Though data is remotely accessible through local Elastic Stores, remote access is expensive. Elastic Executor aligns its workload partitioning with the actual data in its Elastic Store to maximize locality with the migration protocol. During a migration, when worker code running on Elastic Executor asks for a batch of data to process, a local set of data is guaranteed to be returned by keeping track of the keys for local training data.

Dynamic ownership table: In Cruise, workers send requests to specific servers containing the key of the partial model according to each worker's local ownership table. When the ownership update is immediately broadcasted to all worker Elastic Executors during migration, workers can immediately request to the new owner server. For requests that arrive at the old owner server prior to the worker-side ownership update, Elastic Executor of the old owner server refers to its ownership table and redirects

App.	Dataset	Hyper-parameter	Num. of model params
NMF	16x Netflix (1.9M users, 71K movies)	1K rank	1K * 71K
LDA	PubMed (8.2M documents, 141K words)	400 topics	400 * 141K
MLR	Synthetic sparse (100K samples, 160K features)	4K classes	4K * 160K
Lasso			

Table 3.3: Description of datasets used in evaluation.

the requests to the new owning server.

3.4 Evaluation

We implemented Cruise with around 20K lines of code in Java 1.8. We built Cruise on Apache REEF [12], a library for application development on cluster resource managers such as Apache YARN and Apache Mesos. REEF provides a control plane for data processing frameworks including the negotiation with the cluster resource manager and the control channel between containers.

We evaluate Cruise with three machine learning applications. Our evaluation mainly consists of the following four sections: (1) We compare the performance of our expected optimal configuration to that of the actual optimal configuration (§ 3.4.2). (2) We demonstrate that Cruise reduces epoch time, speeding up training (§ 3.4.3). (3) We show how Cruise optimizes the system configuration when resource availability changes (§ 3.4.4) and in heterogeneous environments (§ 3.4.5). (4) We investigate the overhead incurred while optimizing the system. (§ 3.4.6).

3.4.1 Experimental Setup

Default cluster setup: We run experiments on AWS EC2 instances with YARN running on Ubuntu 14.04. Unless explicitly mentioned, we use 32 r4.xlarge instances, each of which has 4 virtual cores, 30.5GB RAM, and 1.25 Gbps network connection [2]. We launch one Elastic Executor per machine to run a worker or a server.

Workloads: We choose three popular ML workloads in different categories: recommendation, classification, and topic modeling as summarized in Table 4.9.

[2]AWS specifies that the r4.xlarge type provides *up to 10 Gbps* network bandwidth. We measured the actual bandwidth with iperf tool.

Non-negative Matrix Factorization (NMF) is commonly used in recommendation systems. The main idea is to find undetermined entries in a given matrix. NMF factorizes a matrix M $(m \times n)$ into factor matrices L $(m \times r)$ and R $(r \times n)$, where $M \approx LR$. We implement NMF via the stochastic gradient descent (SGD) algorithm, similar to the one described in [54]. Matrix R is partitioned across servers, while L is partitioned across workers, where the smallest unit of training data is a single user's rating matrix $(1 \times n)$. NMF experiments use the 16x Netflix dataset whose size is around 40 times greater that the one in the evaluation of [54]. We set mini-batch size to be 10K.

LDA is an algorithm to discover hidden properties (topic) from a group of documents. Each document consists of a bag of words, where LDA associates latent topic assignments. Our LDA implementation uses an efficient variant of the collapsed Gibbs sampling algorithm [61], which is widely used [47, 54]. We run LDA experiments using the PubMed dataset. Our dataset is 15 times larger than one of the datasets used in [54]. We process 1K documents in a mini-batch.

Multinomial Logistic Regression (MLR) is an algorithm for classification. Each d-dimensional observation $x \in R^d$ belongs to one of the M classes, with the model parameter size of $M \times d$. We also implement MLR using SGD. Our experiments use a synthetic dataset generated by a public script from the Petuum framework [7]. The dataset is around 46 times greater that the one used in the evaluation of [54]. We process 1K observations in a mini-batch for our experiments.

Optimizer setup: We observe that the performance of the initial mini-batches fluctuate until the system stabilizes. To prevent Optimizer from computing the cost inaccurately, we configure it to wait until all workers finish a set of mini-batches. In addition, Optimizer does not trigger reconfiguration if the estimated performance gain (in terms of the cost) is below 5% in order to avoid oscillation. As mentioned in § 3.2.3, we use the $O(N)$ analytical solver for the homogeneous environment and use the ILP solver for the heterogeneous environment.

App.	Initial (W, S)	Cruise's Choice	Optimal (W, S)	Rel. Perf
NMF	(27, 5)	(24, 8)	(23, 9)	98.8%
	(18, 14)	(22, 10)		93.9%
MLR	(18, 14)	(26, 6)	(27, 5)	97.8%
	(23, 9)			
LDA	(27, 5)	(18, 14)	(18, 14)	100%
	(23, 9)			
Lasso	(27, 5)	(20, 12)	(20, 12)	100%
	(23, 9)			

Table 3.4: Comparison between the configurations found by Cruise's Optimizer and the ground truth optimum found by the grid search.

3.4.2 Finding Baselines with Grid Search

Before evaluating Cruise's optimization, we find the baseline for all experiments. We simply perform a grid search that runs all possible configurations $(\mathbf{w}, \mathbf{s})$, to find the ground truth optimal configurations for the various experiments. Since such a grid search including $(\mathbf{d}, \mathbf{m})$ is quite complicated, we use heuristics to eliminate these variables. In the homogeneous environment, an even partitioning is intuitively optimal. The result of this grid search yields the 'Optimal (W, S)' column in Table 3.4 for the cases in Table 3.1 3.1. Relative performance is the epoch time in the optimal (W, S), divided by the epoch time in each configuration. In the heterogeneous environment, we distribute blocks proportional to each machine's power based on metrics including worker's local computation time and server's bandwidth.

3.4.3 Optimization in the Homogeneous Environment

After performing the grid search, we run NMF, MLR, Lasso, and LDA each starting with its optimal configuration and the optimal configurations for the other two applications (as the two configurations are reasonable starting points for users running the applications in the same cluster).

Cruise finds configurations close to the ground-truth optimum found in § 3.4.2 for the various cases mentioned in § 3.1 in the homogeneous environment. Table 3.4 shows the comparisons: In NMF and MLR, Cruise chooses near-optimal configurations where the number of machines for each role differs by one node compared to the

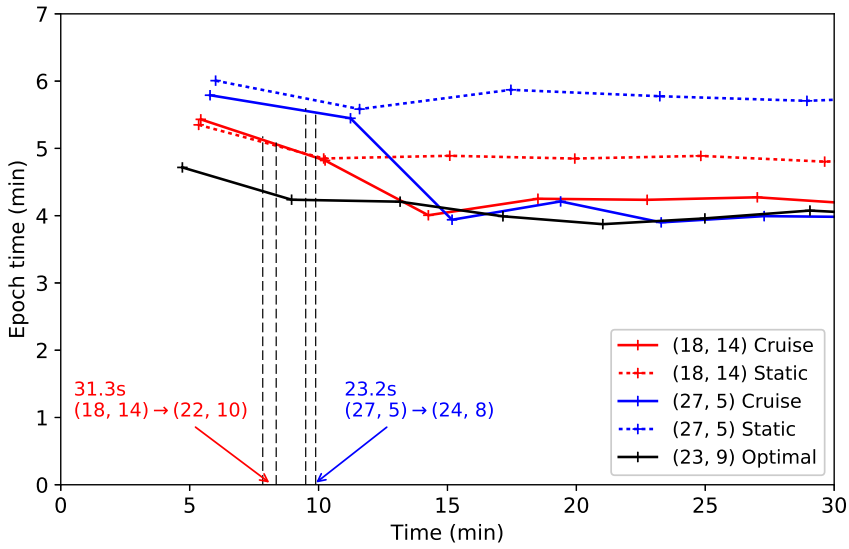


Figure 3.4: Epoch time of an NMF job starting at 3 different configurations.

optimum found by grid search. The resulting performance in terms of epoch time is slightly inferior to the optimum but the difference is smaller than 6.1%. Cruise finds the optimal configuration in LDA and Lasso with the same performance as the optimum.

Figure 3.4 depicts how Cruise decreases the epoch times of NMF. In the figure, the black line shows the global optimum and the blue and red lines show optimizations from the other initial configurations. The dotted lines show the performance without optimization to the corresponding colors. The vertical lines represent the reconfiguration of each cases. Starting at (W:27, S:5), Cruise moves to (W:24, S:8), with the relative performance of 1.1% slower than the optimum at (W:23, S:9). With the initial (W:18, S:14) configuration, Cruise reconfigures to (W:22, S:10), with 6.5% slower performance than the optimum. We observe that Cruise optimizes the misconfigured NMF jobs with significant drops in epoch time of 35.8% and 22.3% in each case, soon stabilizing to that of the new configuration. Our experiments with other applications in the same environment also decrease epoch time by 55.3% and 8.7% in MLR, and 37.5% and 17.4% in LDA, and 42.7% and 28.5% in Lasso.

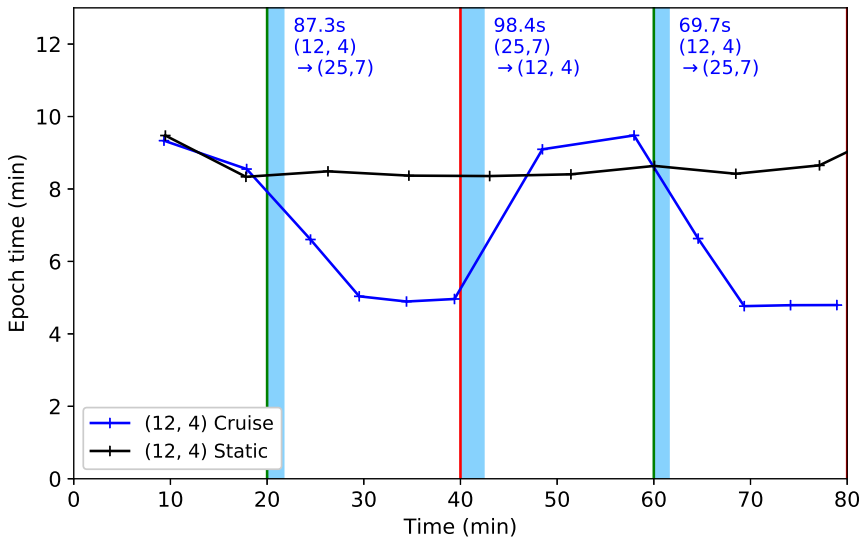


Figure 3.5: Utilizing opportunistic resources in the NMF job.

3.4.4 Utilizing Opportunistic Resources

The previous experiments show how Cruise optimizes system configuration when available resources do not change. Cruise's capability to optimize system configuration during runtime, however, is more powerful when available resources change over time. Cruise keeps track of available resources in the cluster and updates the system configuration if there are changes in resource availability. In this experiment, we assume that the cluster has 16 extra containers that are available opportunistically; starting with 16 containers, we add/reclaim 16 containers at every 20 minutes. We show how Cruise's runtime optimization utilizes opportunistic resources by comparing cases with and without optimization. In both cases, we run ML jobs with the initial configuration of (W:14, S:2), the actual optimum found for 16 containers from experiments.

Figure 3.5 demonstrates that the average epoch time approximately halves when 16 more resources are available. The training time reduces 18.4% and the resource cost reduces 12% by efficiently using oppotunistic resources, which is cheaper than static resources. The cost per hour is 6.7 $\times$ lower for r4.xlarge type. In the figure, the blue line shows Cruise's ability to adapt to resource availability compared to the base-

line drawn in black line, where the configuration is fixed to the initial 16 resources. Vertical lines represent the event of resource addition (green) and reclamation (red). The areas filled in sky-blue denote the reconfigurations. At the 20th minute, the configuration moves toward (W:25, S:7), taking advantage of the added resources. The reconfiguration takes around 87.3 seconds, most of the overhead caused by the data migration of the half of total training data and model data, to the new containers. At the 40th minute, Cruise returns to the previous configuration (W:12, S:4) with 98.4 seconds of reconfiguration overhead for data migration and state cleanup. The additional resources become available again at the 60th minute and Cruise goes to (W:25, S:7), same as the previous optimization at the [20, 40] minutes time interval.

3.4.5 Optimization in the Heterogeneous Environment

There are two different aspects to the setup in the heterogeneous environment from the homogeneous environment. Workload should be partitioned differently corresponding to machine types and each type of machine should be assigned a more proper role (e.g., worker or server). The heterogeneous environment uses two types of machines: in addition to the 28 instances of r4.xlarge, we use 4 faster machines (r4.4xlarge) that have 16 virtual cores, 122 GB RAM, and 5.00 Gbps network connection. [3] In our experiments, we allocate the faster instances evenly to begin with, 2 for workers and 2 for servers. For block partitioning, we start all experiments with *even* partitioning denoted as 'E', whereas the optimal configuration distributes blocks *proportially* to machine's capability, which is denoted as 'P'. For example, we denote a configuration of 20 workers with 3 strong machines and 12 servers with 1 strong machine with even block partitioning as (W:17+3, S:11+1)|E. We run the same three applications with the same starting points as the homogeneous environment, and we show how differently Cruise optimizes the configuration in the heterogeneous environment.

Figure 3.6 shows the results of running NMF starting at (W:25+2, S:3+2)|E. Cruise reconfigures the job to (W:20+4, S:8+0)|E close to the ground truth optimum in the

[3] AWS specifies that the r4.4xlarge type provides *up to 10 Gbps* network bandwidth. We measured the actual bandwidth with iperf tool.

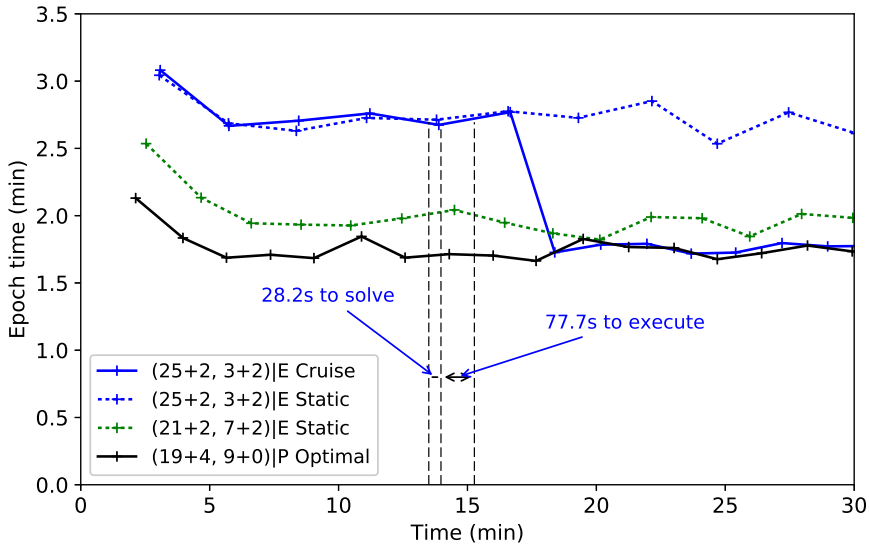


Figure 3.6: Epoch time of NMF at different starting points in the heterogeneous environment.

heterogeneous environment (W:19+4, S:9+0)|P. Data is repartitioned so that the faster instances have about 2 times more blocks than the slower instances, reflecting the heterogeneity. Epoch time decreases by 35.2% (from 162 s to 105 s). Our experiments with other applications in the same environment also reduce the epoch times by 58.3% in MLR starting at (W:16+2, S:12+2)|E and 41.3% in LDA starting at (W:25+2, S:3+2)|E.

To focus on the benefit of role reassignment and workload repartitioning, we run the job again at (W:21+2, S:7+2)|E, without any optimization (the green line). Here, we observe that this static configuration is 12.4% slower (118 s vs. 105 s) than the epoch time for the configuration chosen by Cruise.

3.4.6 Reconfiguration Speed

The optimization procedure is composed of cost calculation and plan execution. Cost calculation takes 30 ms for the homogeneous environment and 37.4 s for the heterogeneous environment on average. More time is spent on plan execution, especially for `move`: the overhead includes (de)serialization time, network transfer time, and the time to acquire the lock on the block to migrate. The data size also affects the time to

execute the operation. The overhead of `add` and `delete` is relatively small, which takes around 2~3 s for resource initialization and cleanup. The time for `switch` is more negligible since there is no cost for resource setup

Here we break down the plan execution of an NMF experiment that starts from (W:18, S:14) in § 3.4.3. The plan changes the configuration to (W:22, S:10), composed of 4 `switch`es from server to worker and 30 `move`s that repartition data and model blocks. It takes 31.3 s in total for our plan executor to execute these operations in parallel. Most of this time is taken during worker-side `move`s, due to the huge size of data being migrated. Input data is divided into 200 worker blocks. A worker block is 100 MB, each of which contains 10K items. 18 `move`s migrate 36 data blocks in total between worker executors. The longest `move` takes 25.8 s, migrating 4 blocks at once, serving as the bottleneck to this plan execution time. On the other hand, model data is divided into 128 server blocks. Block size is 2 MB and each contains only 280 items. The plan migrates 36 model blocks with 12 `move`s. Server-side `move`s take at most 1.4s.

3.5 Related Work

TuPAQ [48] is a system for identifying ML model configurations (e.g., support vector machine vs. logistic regression, hyper-parameter values) that lead to high performance in terms of model accuracy, built on Apache Spark [38, 64]. TuPAQ casts ML model identification as a query planning problem and applies a bandit allocation strategy as well as various optimizations such as batching, optimal cluster sizing, and advanced hyper-parameter tuning techniques to solve the problem efficiently. This study focuses on supervised ML models. In contrast, Cruise addresses the problem of tuning system configuration in the PS architecture.

SystemML [32] is a hybrid runtime system that uses in-memory Control Program (CP) and MapReduce (MR) jobs to run declarative ML programs. There also is another version of the work [6] that applied the same concept to Spark [64], regarding Spark Driver as the CP and Executors as the worker jobs. The system focuses on optimizing

memory configurations during runtime when there is a change in available resources. On the other hand, Cruise optimizes the running time of ML applications on the PS architecture by automatically tuning system configuration.

Yan et al. [60] propose a cost formulization that predicts the computation and communication overheads of Deep Neural Network (DNN) applications by modeling the internals of the algorithm. In contrast, Cruise measures computation and communication time at run time instead of modeling the internals of the algorithm, uses the run-time measurement for cost-based optimization, and applies the estimated optimal system configuration by reconfiguring a running job, which allows us to take advantage of resource elasticity.

Starfish [29], built on Apache Hadoop [51], performs optimization on MapReduce. It gathers job profiles from runtime statistics via dynamic instrumentation for job-level tuning, guaranteeing shorter execution times. Many of Starfish's design considerations come from the MapReduce programming model, while Cruise targets ML applications running on the PS architecture.

Recent works like Bösen [54], Ako [53], and MALT [39] do not decide on the number of workers and servers since each node runs both worker and server. They have different styles of model synchronization. Bösen is a PS implementation, which requires all-to-all communication. Ako is a peer-to-peer DNN training system. Ako exchanges partial gradients across multiple rounds to adjust the hardware and statistical efficiency. Similarly, MALT is a peer-to-peer ML training system where each node exchanges parameter updates with $\log n$ nodes deterministically. In contrast, Cruise employs cost-based optimization, allows flexible worker and server allocation, handles elastically changing resources, and considers heterogeneous environments.

3.6 Summary

In this chapter, we present a methodology to automatically tune system configuration of PS-based ML systems. We build Cruise by extending an existing PS-based system to optimize system configuration based on the methodology. Cruise Optimizer estimates

the optimal system configuration - resource configuration and workload partitioning - using a cost-based model with runtime metrics. Elastic Runtime enables efficient runtime reconfigurations according to the computed optimal configuration. Our evaluation shows that Cruise frees ML application developers of choosing right system configuration by tuning system configuration automatically. Cruise is publicly available at `https://github.com/snuspl/cruise`.

Chapter 4

A Scheduling Framework Optimized for Multiple Distributed Machine Learning Jobs

Machine Learning (ML) is growing ever more popular with the diversity of its use cases. However, distributed ML training jobs often suffer from inefficient resource usages, only using up to about half of the overall CPU and network resources that they are provided with, due to the fact that ML jobs typically consist of iterative training cycles that repeat computation and communication steps, each of which uses only a single type of resource intensively. This results in low resource utilization rates, as it leaves most of the other resource types idle during each step. Although asynchronous execution of the different steps or co-locating multiple jobs with naive trial-and-error have mitigated the problem to a certain degree, they often suffer from the model staleness and the interference between the co-located jobs.

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. Harmony enables fine-grained execution by first dividing each job into subtasks, each of

which dominantly uses a single type of resource. Then, Harmony models the performance of co-located jobs based on runtime metrics to identify the jobs that can run harmoniously, and dynamically groups them in real-time for higher utilization rates. While the increased number of simultaneous jobs increases the memory pressure, Harmony resolves the problem by using 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.

The rest of the chapter is organized as follows: §4.1 describes the problem that we aim to solve, §4.2 illustrates the overview of Harmony, §4.3 describes how Harmony divides jobs into subtasks and enables multiplexing of multiple jobs while mitigating memory pressures, §4.3.2 elaborates on how Harmony profiles and models jobs to predict and appropriately group jobs to co-locate on the provided nodes, §4.4 presents the evaluation results and the comparison between Harmony and the baselines that represent existing systems, §4.5 discusses the limitations of Harmony and the future works, §4.6 covers related works, and §4.7 concludes.

4.1 Resource Under-utilization Problems in PS ML Training

In a training iteration, each step (e.g., PULL, COMP, PUSH) 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 4.1 shows how CPU and network resources are underutilized while running ML applications with different algorithms, different hyper-parameters, and with different datasets. 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 us-

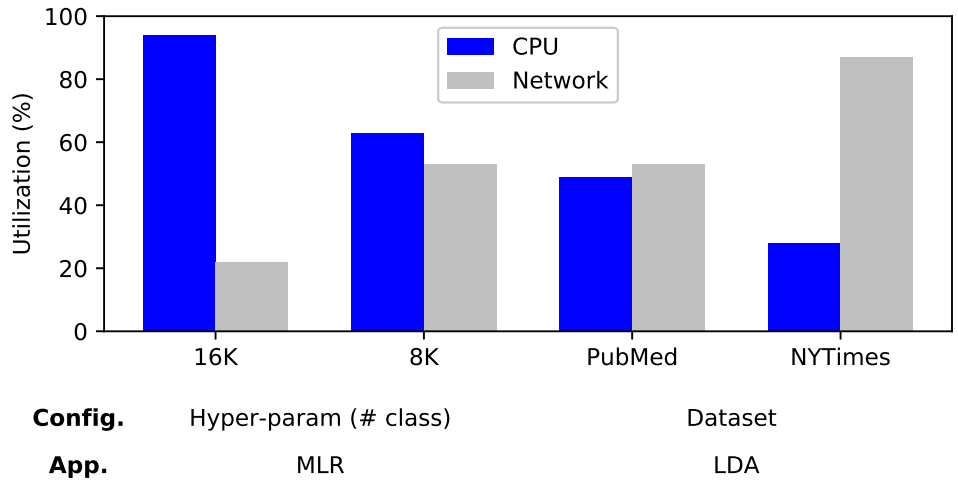
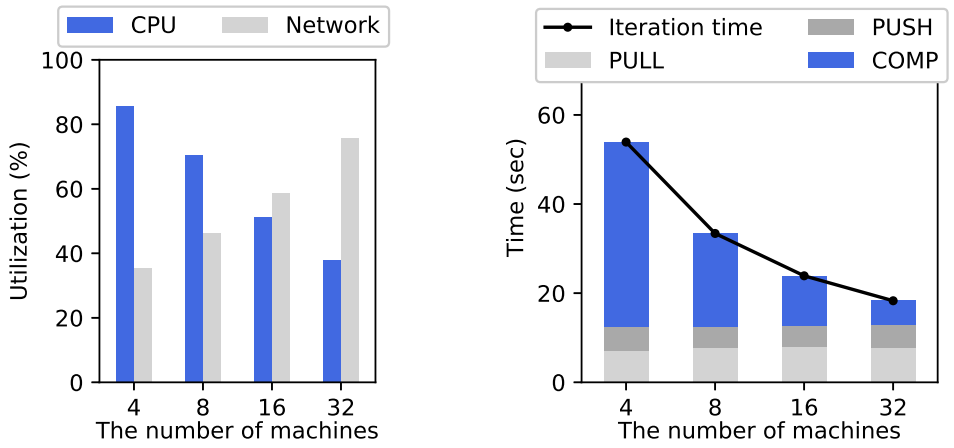


Figure 4.1: Resource utilization of maching learning training, varying applications, datasets, and hyper-parameters.

ing our PS system, which has comparable performance to an open-source PS system referenced in §4.4. 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 4.2 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 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.

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



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

Figure 4.2: Running a job with different number of machines.

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 4.3, we empirically show how co-located parameter server jobs may still lead to resource under-utilization. In addition to the MLR application used in Figure 4.1, 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 4.3. Nevertheless, the overall utilization rates do not improve much even when co-located with other workloads, as shown in the right half of Figure 4.3. We first list the observations of each of the cases as follows:

- **NMF+Lasso**: Although NMF shows relatively high CPU utilization and Lasso shows relatively high network utilization when run on their own, co-locating two jobs do 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

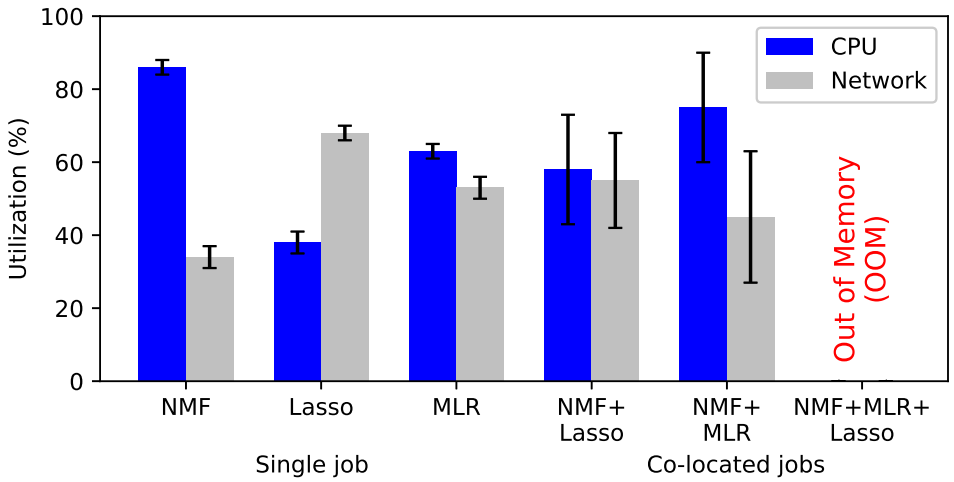


Figure 4.3: Resource utilization of co-located multiple parameter server jobs.

larger compared to when running each job on their own, which indicates that resource utilizations are more unpredictable.

- **NMF+MLR**: Similar to the NMF+Lasso case, this co-location also results in an average utilization of different resources and high variance in the utilization rates, instead of in higher utilization rates.

- **NMF+MLR+Lasso**: 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 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 4.4a. 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

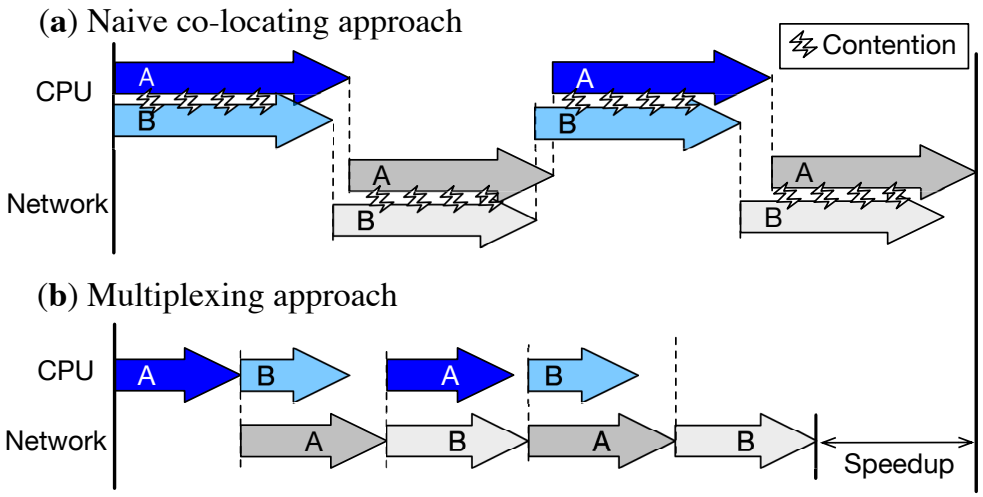


Figure 4.4: Comparison of job scheduling approaches. The figure illustrates only two iterations of jobs for simplicity.

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. As a result, arbitrarily executing multiple co-located jobs incurs higher memory pressure.

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.

4.2 Harmony Overview

We introduce Harmony, a new scheduling framework that embodies our approach, which co-locates jobs with complementary 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 4.4b. 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 4.5. 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

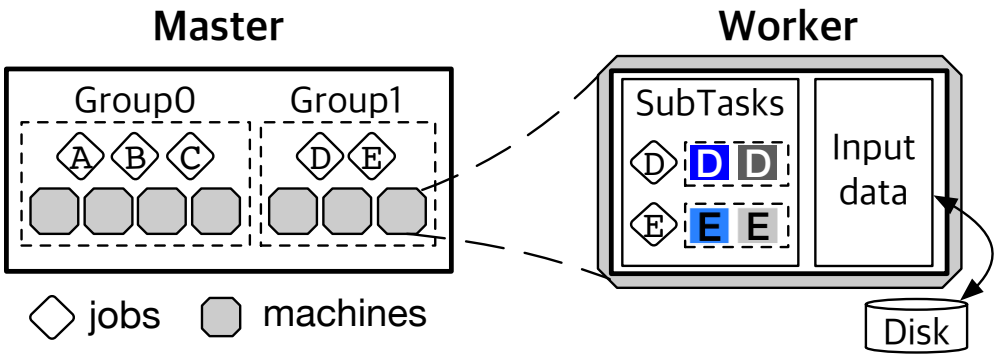


Figure 4.5: Harmony scheduling overview.

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.

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 §4.3.2, 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`).

4.3 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.

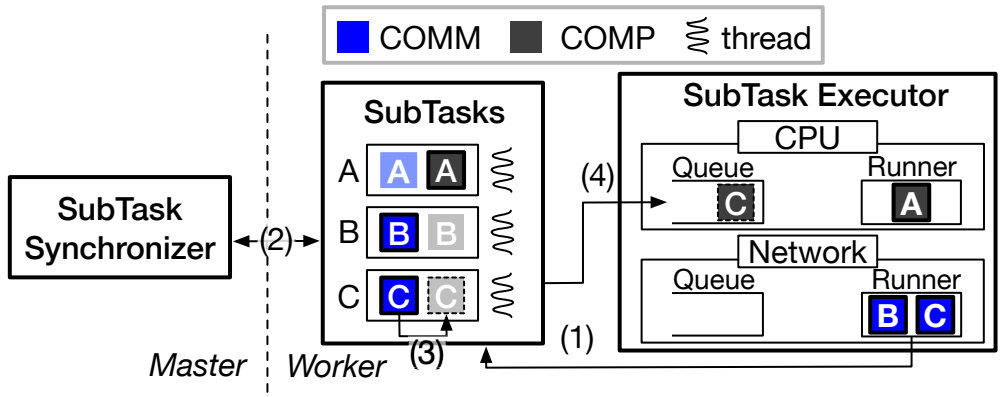


Figure 4.6: Scheduling and execution of jobs A, B, and C, where A is at the COMP subtask, and B and C are at the COMM subtask.

4.3.1 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 4.6 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 that uses the network resources whenever available, while yielding the network resources to the primary network subtask whenever a contention occurs.

Subtask scheduling and execution is illustrated by an example shown in Figure 4.6(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.

4.3.2 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*.

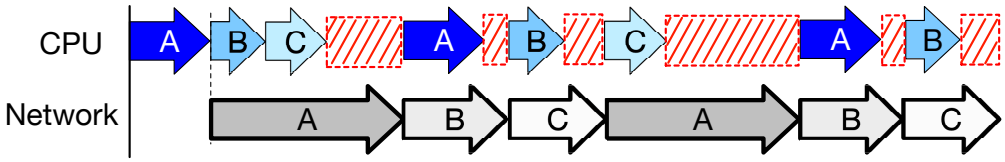
A job group should have the balanced use of resources and the allocated resources should be keep busy during group iterations. To achieve this goal, Harmony makes a scheduling decision using runtime metrics and the performance model.

In this section, we describe how Harmony collects runtime metrics (§4.3.2) and models performance of co-located jobs based on the collected metrics (§4.3.2), and finally schedules jobs and machine resources (§4.3.2) and performs regrouping during runtime (§4.3.2).

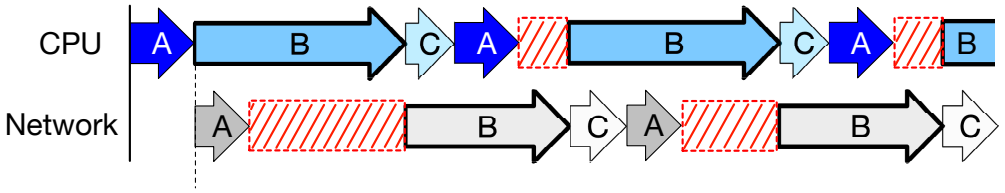
Profiling

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. Resource utilization can be profiled using the execution time of the individual subtasks. ML jobs on Harmony show stable performance with reduced resource contention and thus the profiled metrics of subtasks can be meaningfully reused, while being updated using moving averages for up-to-date information. Group iteration time and the resource utilization can be predicted using these metrics.

Harmony has no information about the jobs at submission. Harmony first tries to run as many jobs as possible at initialization, naively grouping and allocating the smallest number of machines for each job, to minimize the communication overhead and get information from as many jobs as possible. A 3-tuple is produced after each job j in group g is profiled, which consists of the average execution times of CPU and Network subtasks and the number of machines allocated to the group when collecting metrics $(T_{cpu_j}, T_{net_j}, c_g)$. Although multiple jobs are co-located on the machines, we can obtain meaningful metrics for each subtask, since individual subtasks are executed in isolation from other subtasks. After the profiling, the jobs carry on with their execution, until they are later paused or migrated to other machines for better co-location with other jobs. The initial profiling phase terminates when the profiling for all initially started jobs completes. Those not profiled yet wait in the waiting queue and are selected to profile later.



(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.

Figure 4.7: Problematic cases of unbalanced co-located jobs.

For jobs not selected in the initial profiling and jobs arrived later, 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 U.

Performance Modeling

When predicting group iteration time using the collected metrics, Harmony considers several cases of non-uniform resource use of jobs into account, since naively grouping jobs usually does not produce results where multiple co-located jobs fully utilize the resources. Figure 4.7 shows two cases where naive subtask scheduling can be problematic. In the figure, subtasks with bold lines incur under-utilization of resources and Red hatched boxes represent idle resources. First, Figure 4.7a 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 4.7b 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:

$$T_{g\_itr_g} = \max\Big(\sum_{j \in g} T_{cpu_j}, \sum_{j \in g} T_{net_j}, \max_{j \in g} T_{j\_itr_j}\Big) \tag{4.1}$$

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:

$$T_{cpu_j} \propto \frac{1}{m_g} (j \in g) \tag{4.2}$$

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.4.1.

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:

$$U(g) = \begin{bmatrix} U_{cpu_g} & U_{net_g} \end{bmatrix} = \begin{bmatrix} \dfrac{\sum_{j \in g} T_{cpu_j}}{T_{g\_itr_g}} & \dfrac{\sum_{j \in g} T_{net_j}}{T_{g\_itr_g}} \end{bmatrix} \tag{4.3}$$

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 sum of the utilization rates of all job groups, where G is the set of job groups:

$$U = \frac{\sum_{g \in G} (m_g \times U(g))}{\sum_{g \in G} m_g} \tag{4.4}$$

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.

Grouping Jobs and Allocating Machines

Based on the model, Harmony makes a scheduling decision that groups jobs and allocates machines to each job group. Concretely, we need to address the following questions:

- How many jobs should we run concurrently?
- How many job groups should we create, and how should we assign the jobs to the groups?
- How should we allocate resources across the job groups?

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 Section 4.4.

Algorithm 1 presents the scheduling algorithm. The algorithm 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

Algorithm 1: Job scheduling algorithm.

input : J_{paused}: list of paused jobs,
$J_{profiled}$: list of profiled 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 $J_{to\_sched} \leftarrow J_{profiled} \cup J_{paused} \cup J_{running}$
3 $U_{max} \leftarrow 0$
4 **for** $n_j \leftarrow 2$ **to** $|J_{to\_sched}|$ **do**
5 $J_{to\_group} \leftarrow J_{to\_sched}[0 : n_j - 1]$
6 ${n_G}^* \leftarrow \underset{n_G}{\operatorname{argmin}} \sum_{j \in J_{to\_group}} |T_{cpu_j}(n_G) - T_{net_j}|$
7 $G_J \leftarrow$ `assignJobs`(J_{to_group}, n_G^*)
8 $G_M \leftarrow$ `allocateMachines`(G_J, M, n_G^*)
9 $G \leftarrow (G_J, G_M)$
10 **if** $U(G) > U_{max}$ **then**
11 $U_{max} \leftarrow U(G)$
12 **else**
13 `break`
14 **return** G

number of machines and thus the same DoP for all of the jobs (L6). Here, since we assume that the DoP is equal among the job groups, $m_g \propto \frac{1}{n_G}$ and thus $T_{cpu} \propto n_G$ ($\because Eq.4.2$). 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).

Algorithm 2 describes the grouping algorithm in more detail, which assigns jobs J into a given number of groups n_G^*, towards a higher resource utilization. In order to prevent job-bound cases and to minimize the average group iteration time of the job groups, 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, then it would result in a higher average group iteration time, due to the scattered large jobs, so we try to keep the large ones together. In order to do this, the scheduler first sorts jobs by their job

Algorithm 2: Assign jobs to groups.

input : J: jobs to assign, n_G: number of groups.

output: G_J: list of job groups.

```
1  Function assignJobs(J, n_G):
       // Sort J by T_{j_itr} in descending order.
2      J_sort ← sortByDsc(J, j ⇒ T_{j_itr_j})
3      G_J ← ∅ // G_J: list of job groups.
4      for i ← 0 to n_G − 1 do
5          g ← ∅ // g: i-th job group.
6          while |g| ≤ ⌊|J| / n_G⌋ do
               // l.popFirst(pred): pop the first item in l, which
                  satisfies pred.
               // T_{diff_j} : T_{cpu_j} − T_{net_j}, S_{diff_g} : Σ_{j∈g}(T_{diff_j})
7              j ← J_sort.popFirst(j ⇒ T_{diff_j} × S_{diff_g} ≤ 0)
8              if j = ∅ then
9                  j ← J_sort.pop()
10             g.add(j)
11         G_J[i] ← g
12     J_sort' ← sortByDsc(J_sort, j ⇒ |T_{diff_j}|)
13     foreach j ∈ J_sort' do
           // g*: group that has the most opposite resource use from
              j.
14         g* ← argmin_{g∈G_J} |T_{diff_j} + S_{diff_g}|
15         G⁻ ← ∅ // G⁻: groups to exclude
16         while T_{j_itr_j} > max_{j'∈g*} (T_{j_itr_{j'}}) do
17             if isJobBound(g* ∪ {j}) then
18                 G⁻.add(g*)
19                 g* ← argmin_{g∈(G_J − G⁻)} |T_{diff_j} + S_{diff_g}|
20             else
21                 break
22         g*.add(j)
23     return G_J
```

iteration time $T_{j\_itr_j}$ (L2).

Then, it determines whether the jobs of the group g are CPU-bound ($\sum_{j' \in g} T_{diff_{j'}} = \sum_{j' \in g}(T_{cpu_{j'}} - T_{net_{j'}}) > 0$) or network-bound ($\sum_{j' \in g} T_{diff_{j'}} < 0$), and chooses the largest job j that has the complementary resource-bound pattern to place in the group ($T_{diff_j} \times \sum_{j' \in g} T_{diff_{j'}} < 0$) (L7). If there is no job j with a complementary resource-bound pattern, it just picks the largest one out of the jobs J_{sort} (L8-9). This process repeats until each of the n_G^* job groups are filled up with $\lfloor \frac{|J|}{n_G} \rfloor$ jobs (L6).

For the remaining $|J| - n_G \cdot \lfloor \frac{|J|}{n_G} \rfloor$ jobs, the scheduler sorts the jobs by the "resource-imbalance" of the job $|T_{diff_j}|$ (L12), and finds the group g^* that would balance out the resource use the most in that order ($g^* \leftarrow \text{argmin}_{g \in G_J} |T_{diff_j} + \sum_{j' \in g} T_{diff_{j'}}|$) (L14). In the case where the iteration time of job j is larger than all of the jobs in the chosen job group g^* ($T_{j\_itr_j} > \max_{j' \in g^*} T_{j\_itr_{j'}}$), it checks if a new job group $g^{**} \leftarrow g^* \cup j$ would be job-bound ($T_{g\_itr_{g^{**}}} = T_{j\_itr_j}, \because Eq.4.1$), and looks for the next best group if that is the case ($g^* \leftarrow \text{argmin}_{g \in (G_J - \{g^*\})} |T_{diff_j} + \sum_{j' \in g} T_{diff_{j'}}|$), until it is no longer job-bound by the job j (L15-22).

Lastly, the algorithm fine-tunes the result by swapping jobs between the groups, until the convergence of the job models. First, it picks the most imbalanced group $g_u \leftarrow \text{argmax}_{g \in G_J} |\sum_{j \in g} T_{diff_j}|$, and finds the group that has the most complementary resource use $\text{argmax}_{g \in (G_J - \{g_u\})} |\sum_{j \in g_u} T_{diff_j} - \sum_{j \in g} T_{diff_j}|$. 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 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 ($\because Eq.4.2$), reducing the CPU-bound cases (Eq.4.1).

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 to get its statistics as described

in § 4.3.2. 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 §4.4.6. 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.

4.3.3 Dynamic Data Reloading

When running a single job on dedicated resources, memory pressure rarely becomes a problem. However, as Harmony runs multiple jobs simultaneously, higher memory pressure is inevitable due to the increased number of concurrent jobs. Moreover, 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. Data blocks can be kept in disk (disk-block) and dynamically reloaded in the background, while blocks in memory (memory-block) are processed by the corresponding subtasks as illustrated in Figure 4.8. Each disk-block should be loaded before its use and is spilled right after the use.

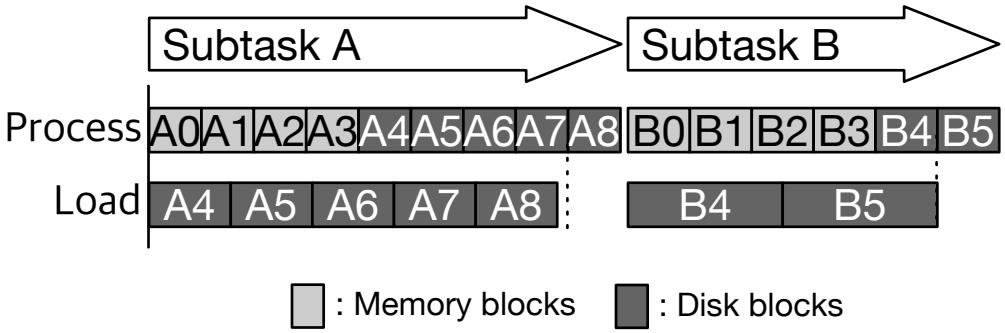


Figure 4.8: Dynamic data reloading in background.

An important factor we need to decide is the ratio of disk-blocks out of the total number of input data blocks. 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. Naturally, $1 - \alpha_j$ is the ratio of memory-blocks. If block loading speed is slower than the speed of block processing speed, α_j should be small enough for reloading not to block task progress. We can determine the maximum bound of α_j with $T_{b\_proc_j}(B_{total_j} - 1) \geq T_{b\_load_j}(B_{disk_j} - 1)$, where $T_{b\_proc_j}$ is the time that it requires for a single data block to be processed in memory and $T_{b\_load_j}$ is the time that it requires Harmony to load a single disk-block.

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.

When input data spilling is not enough for relieving memory pressure (when α already becomes 1), Harmony also enables spill/reload of model data. However, model data provides less benefits due to the following reasons. First, a subtask may access the whole model data and corresponding disk-blocks should be loaded before the start of a subtask. It's different from input data, whose data element is independent from each other and data loading and processing can be pipelined by streaming data blocks.

Second, when spilling, model data should be checkpointed because it is mutable. For these reasons, Harmony rarely enables spill/reload of model data, only if memory pressure is still high after aggressively increasing the disk-side portion of input data. With experiments, we found that input data handling is sufficient at most of the times, but in several cases model data handling successfully reduces memory pressure and avoids failures by OOM.

4.4 Evaluation

We have implemented Harmony with 8.8K lines of Java code. We have built Harmony on top of Apache REEF [13,55] that provides common functionalities such as the master and worker abstractions, which are required to write systems running in distributed environments.

In this section, we try to prove two statements about Harmony with the evaluation results to quantify its effectiveness against other scheduling approaches:

- Harmony provides shorter average JCTs and makespans with its scheduling approach for diverse workloads compared to existing approaches.
- Harmony provides a scheduling algorithm that is scalable enough to schedule large-scale workloads within a reasonable time.

4.4.1 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, while reducing the network overheads that occur with lower DoP and higher CPU utilization. Existing works that take similar approaches for allocating resources to each job include Optimus [45] and SLAQ [65].

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

Apps	Domain	Dataset	Input (in GBs)	Model (in GBs)
Non-negative Matrix Factorization (NMF)	Recommendation	Netflix64x [4]	45.6	1.0
		Netflix128x	91.2	5.0
Latent Dirichlet Allocation (LDA)	Topic modeling	PubMed [22]	4.3	2.1
		NyTimes [22]	0.6	1.1
Multinomial Logistic Regression (MLR)	Classification	Synthetic [8]	78.4, 155.0	12.0, 24.0
Lasso	Regression			

Table 4.1: Workloads used for evaluation. In MLR and Lasso, we use a script for generating synthetic datasets, which is included in Bösen.

different allocations of resources cause greater variance in the makespan compared to the isolated baseline. To find the optimal solution of the approach, we run all possible cases in a brute-force manner, and report the best and the worst case among the possible choices. This baseline represents the approach introduced in Gandiva [58], which does not provide fine coordination between co-located jobs and an analytical basis for job grouping. In our baseline, the minor optimizations described in Gandiva for finding better match of jobs are neglected, as we present the best choice obtained from the exhaustive search.

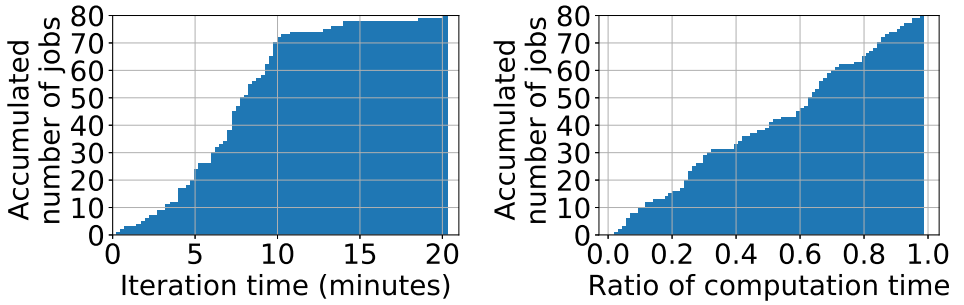
As the baseline systems mentioned above are not open-sourced, we implement their scheduling schemes on Harmony.

4.4.2 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 with the same specifications is used as the master.

As specified in Table 4.1, we use 4 applications, 2 datasets, and 10 different hyper-parameters for each of the 80 different (application, dataset) tuple. The distribution of workload characteristics such as the iteration time and the computation to communication ratio are illustrated in Figure 4.9. We measure the statistics using DoP 16 for all jobs.

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



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

Figure 4.9: Key characteristics of workload used for evaluation. We use DoP 16 for these experiments.

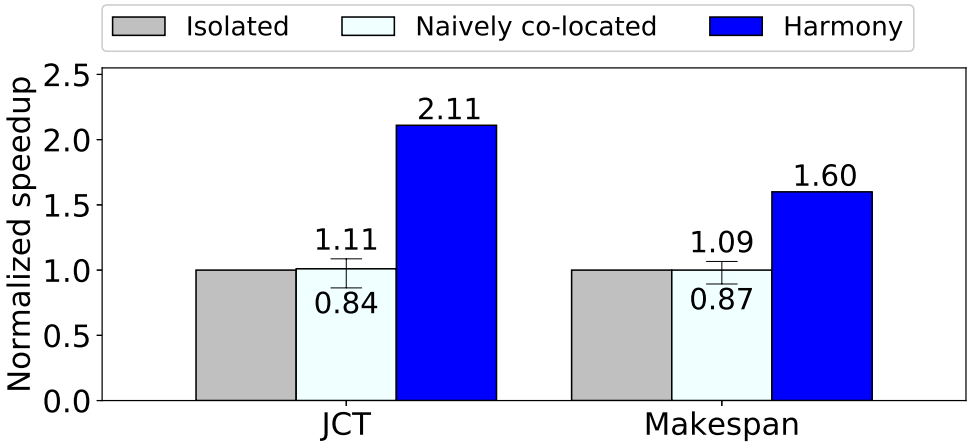


Figure 4.10: JCT and makespan in Harmony and the baseline approaches.

epoch and determine the convergence by comparing the objective value with the predefined 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 [8], an open-source PS system, with its staleness parameter set to 0 for synchronous training. With this condition set, we compare the performances of the scheduling methods in our main evaluations.

4.4.3 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 4.10, where the makespan and JCT are normalized by the baseline *isolated* approach and for *naively co-located* approach, the bar means average value and the error bar represents max/min values. 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 4.11 shows that Harmony shows higher and less fluctuating resource utilization patterns. The vertical lines represent the completion time for all jobs (i.e., makespan). 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 average, while divided into 6.7 job groups across all 100 machines.

4.4.4 Performance Breakdown

Figure 4.12 presents how the individual techniques of Harmony contribute to the overall performance benefit. We compare the performance by gradually adding the different

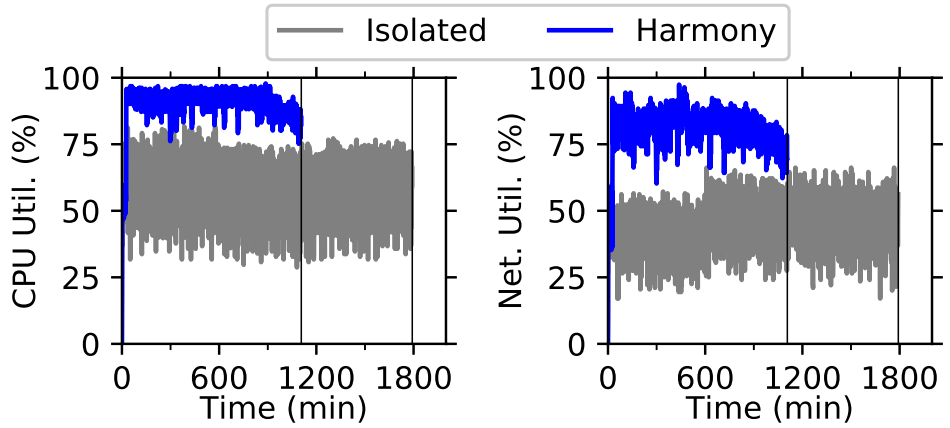


Figure 4.11: Resource utilization of Harmony and isolated-approach during an experiment that runs 80 jobs.

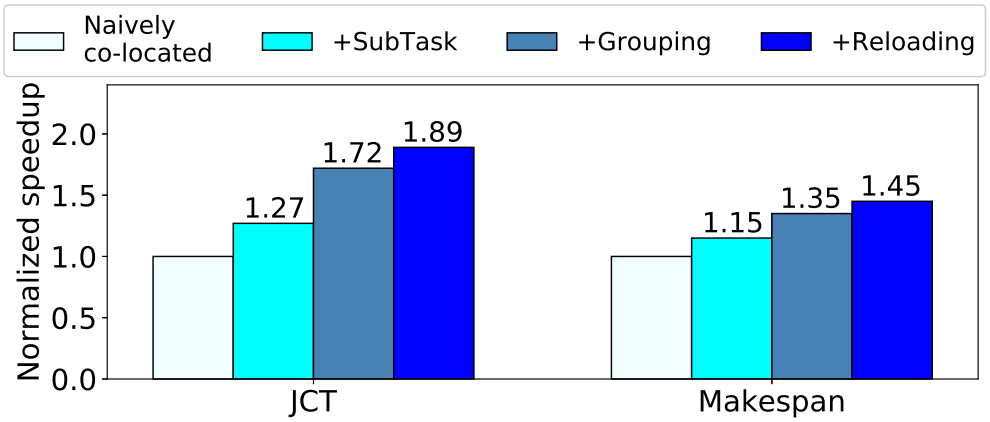


Figure 4.12: Breakdown of performance gain.

techniques on top of each other. For *naively co-located* approach, we use the value of the best case.

Subtask: With subtasks, we achieve 1.27× and 1.15× speedup in JCT and makespan, respectively. Subtasks eliminate and minimize the contention, resulting in better resource utilizations even when the group runs in the 'naively co-located' fashion.

Grouping: We perform scheduling algorithms mentioned in §4.3.2 for grouping jobs together. After considering the cases for different sets of job groups, we achieve 1.72× and 1.35× speedup in JCT and makespan, respectively. Note that the JCT improvement is larger, since we prefer the smallest number of jobs as possible while increasing the utilization in our grouping algorithm, which avoids sacrificing the

makespan, differently from the *isolated* approach.

Reloading: Dynamic reloading (§4.3.3) enables more number of jobs to run simultaneously. Previous experiments limit the number of jobs in a job group so that the amount of the required memory does not exceed the sum of the machines' memory assigned to the group, as OOM occurs otherwise. More number of concurrent jobs contributes to the speedup, since it results in a higher resource utilization, compared to the case where it was not possible due to the memory limit.

4.4.5 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 4.9b). As a result, the two set of jobs are relatively computation-heavy and communication-heavy compared to the base workload.

Figure 4.13 shows the comparison result. 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 based on the information extracted from grouping decisions of the

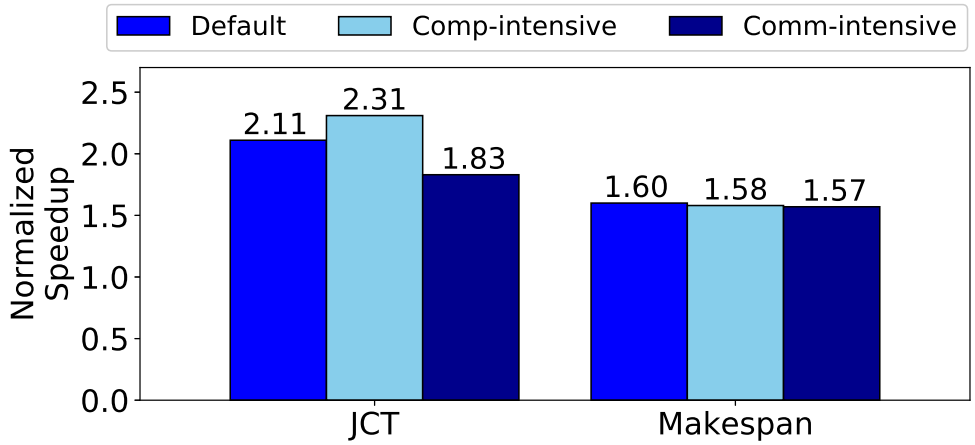


Figure 4.13: JCT and makespan of workloads with different resource usage ratios.

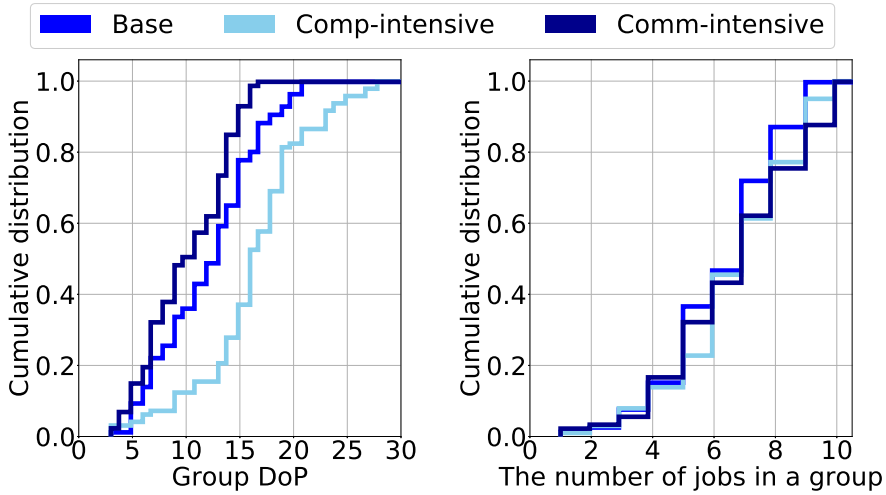


Figure 4.14: Distribution of group DoPs and the number of jobs in a group.

scheduler during whole execution, Harmony uses larger DoPs for the computation-intensive workload and smaller DoPs for communication-intensive workload as illustrated in the left graph of Figure 4.14. 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 4.14.

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, varying the arrival rates as illustrated in Figure 4.15.

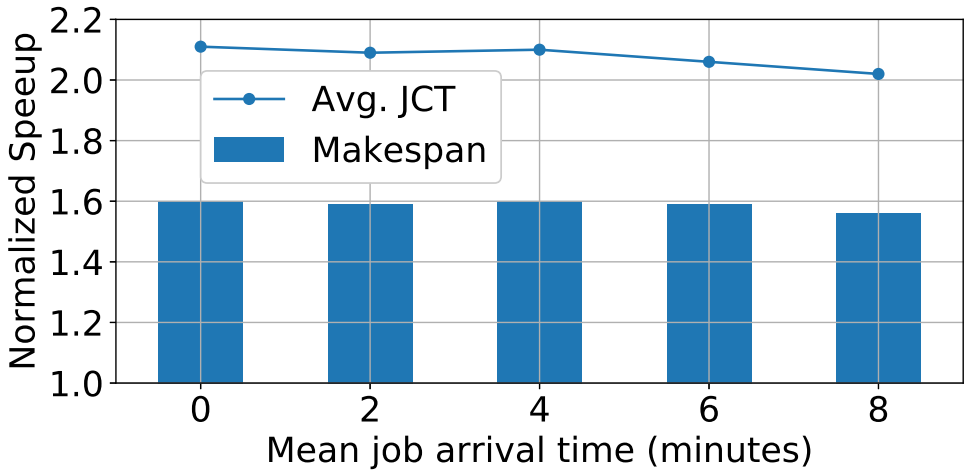


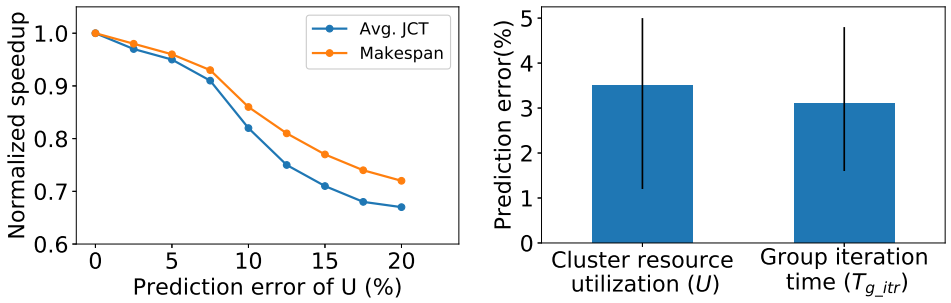
Figure 4.15: Performance varying job arrival rate. 0 arrival time means that we submit all jobs at once.

As the mean job arrival time increases, the performance slightly decreases as the number of concurrent jobs decreases, preventing them from being fully multiplexed at all times. In contrast, when the jobs are submitted more frequently, the performance increases with better multiplexing. The effect of multiplexing does not increase more when the mean job arrival time is less than 4 minutes, because at that point Harmony already has enough number of jobs to multiplex.

Lastly, we use job arrival rates processes from Google cluster workload traces [56]. 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.

4.4.6 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 4.16a 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.



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

Figure 4.16: Accuracy of performance model. The vertical line represents the min/max values.

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 §4.4. Thanks to subtask execution model, the prediction error stays below 5% at all times as illustrated in Figure 4.16b.

4.4.7 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 4.17 shows the comparison result of resource utilization, average JCT, and makespan, between the solution found with the exhaustive search and the one provided by 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 as we show below.

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

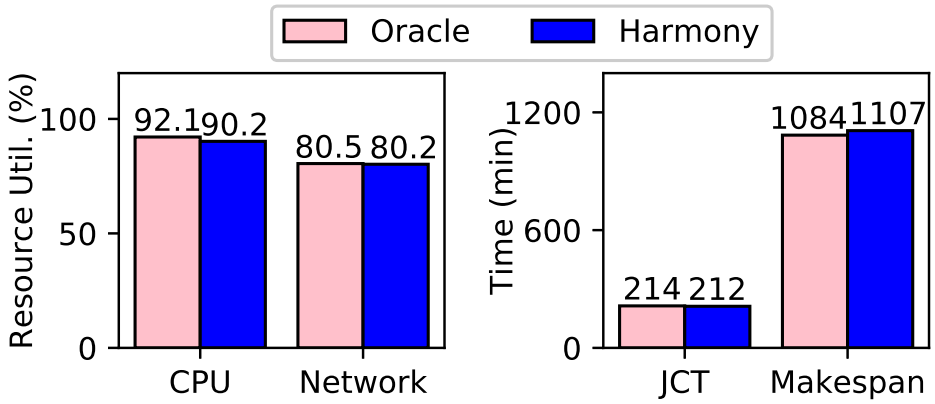


Figure 4.17: Comparison of resource utilization, average JCT, and makespan to exhaustive search (Oracle).

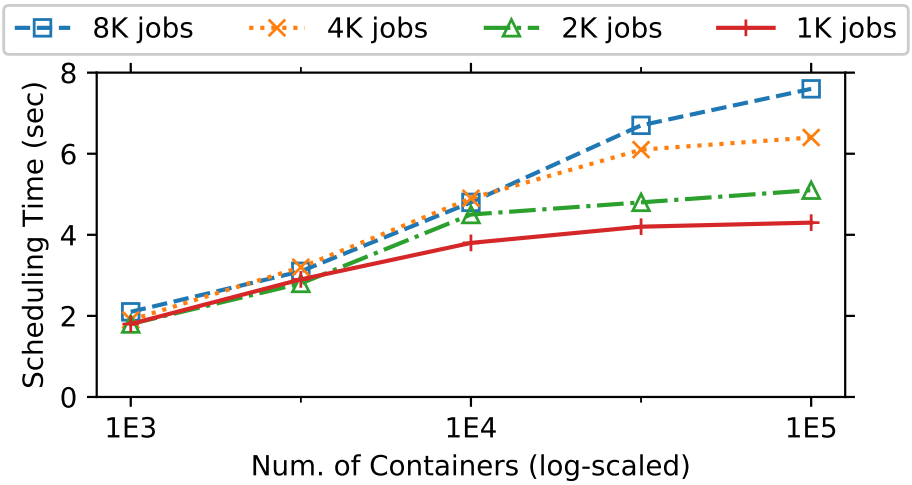


Figure 4.18: Scalability varying the number of machines and jobs to schedule.

is 1.2 seconds in Harmony and 13.8 minutes in Oracle obtained by exhaustive search, respectively. To test on a large-scale environment (e.g., datacenters), we emulate the submission and scheduling of thousands of jobs to thousands of machines.

As illustrated in Figure 4.18, Harmony can schedule 8K jobs to 10K machines within 5 seconds. This result is comparable to the performance of a scheduler developed recently [45] and a default scheduler of a general RM [57]. Optimus is scalable up to 4K jobs to 16K machines within 5 seconds. On the other hand, the exhaustive search algorithm for 4K jobs on 10K machines takes about 10 hours, due to the expo-

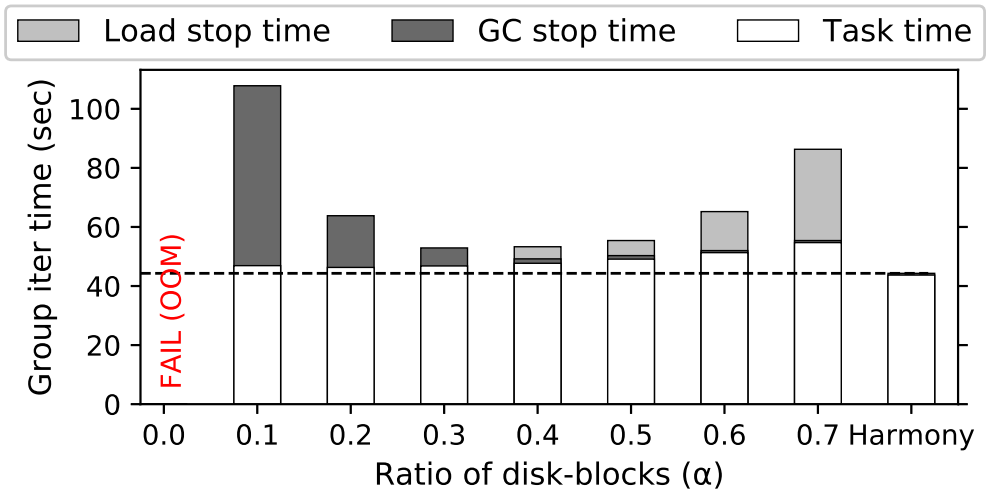


Figure 4.19: Proportion of fixed block ratio to our dynamic adaptation. Load stop time and GC stop time means the time the job tasks are stopped due to data reloading and GC, respectively.

nential growth of the time for running the scheduling algorithm with more number of jobs and machines.

4.4.8 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. Figure 4.19 illustrates the result. When α is too high, group iteration slows down due to the load stop time, which means 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 §4.4.3, 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, Harmony automatically en-

ables 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) from occurring.

4.5 Discussion

Regrouping costs: We have tried to minimize the overhead caused by regrouping, and the overhead is trivial compared to the actual job execution in our experiments. In future work, we may estimate the cost and the benefit from the regrouping and decide whether to apply the new decision or not. To achieve this, we can use convergence prediction techniques introduced in works like SLAQ [65] to predict the time for the job completions, which will make the present scheduling decision sub-optimal.

Execution of small number of jobs: As shown in the evaluations, Harmony is less effective with a smaller number of jobs. In order to reduce such cases, we could use convergence prediction techniques [65] to balance out the workloads so that sufficient numbers of jobs finish at a similar time. Otherwise, we could use a hybrid approach with asynchronous training. When we encounter a case where the number of jobs is not enough to multiplex, we could switch to asynchronous training mode to utilize the idle resources.

Fault tolerance: 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. Although each job could recover by restarting from its latest checkpoint, we may share memory across the jobs, which may be a bit tricky to use, but more safe from cascading failures in future work.

4.6 Related Work

Schedulers specialized for multiple ML jobs: While many organizations actively use general-purpose schedulers (e.g., DRF scheduler [24], Hadoop fair and capac-

ity scheduler [63]) for scheduling ML jobs in production [5, 17, 33], many recent researches have introduced scheduling solutions specialized to ML workloads. Optimus [45] minimizes the training time of multiple DL/ML jobs in GPU/CPU servers based on online "resource-performance models". Oasis [3] focuses on maximizing parameter server-side utilization using online algorithm for scheduling jobs and allocating resources for ML jobs. SLAQ [65] dynamically schedules resources between jobs to maximize system-wide quality improvement based on quality prediction model, but they neglect network-intensive workloads, causing inefficiency in using network resources. Dorm [50] dynamically partitions a cluster and allocates resource partitions to jobs for resource efficiency and fairness. Tiresias [25] schedules DL jobs focusing on characteristics of unpredictable execution time and all-or-nothing execution model. All of the above works have greatly improved the cluster performance, but none of them considers 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.

Schedulers co-locating multiple jobs: Recently, Gandiva [58] has been suggested to support co-location of DL jobs into the same GPU with its 'packing' mechanism, but its main goal is in providing early feedback for hyper-parameter searching. Unlike Harmony, Gandiva treats each job as black-box and do not coordinate and control the task executions in workers, and results in interferences and limited performance gain or loss in some cases. Harmony focuses on improving the utilization of both the CPU and network resources for multiple co-located ML jobs by coordinating task executions in a fine-grained manner. Zhang et al. [66] solves the resource under-utilization problem in datacenters by co-locating batch jobs and latency-sensitive jobs. They harvest spare CPU cycles and disk spaces from latency-sensitive jobs, which typically over-provision resources, and run batch jobs with lower priority. Unlike Harmony, they handle two different types of workloads that have different schedule priorities. Optimus [45] also considers co-location of multiple workers and servers in a machine and handles communication overhead between them. However, it assumes isolation of resources within a machine (e.g., container) between the co-located workers and

servers.

Dynamic resource scheduling for ML: Instead of using static allocation of resources [30, 52], recent works introduce resource elasticity in distributed ML frameworks. Optimus [45] dynamically reallocates resources between jobs or between workers and servers, Gandiva [58] dynamically migrates jobs between GPUs for less interference, or between GPU and CPU for time-slicing, and Dorm [50] dynamically partitions resources to isolated jobs to increase the resource utilization. Litz [46] and Proteus [28] migrate jobs in execution between reliable instances and transient instances, for better cost efficiency in cloud environment. Similarly, Harmony dynamically migrates jobs and reallocates resources for better performance. The difference is that Harmony designs scheduling algorithm and job migration protocol to be simple but effective for multiple co-located jobs to minimize regrouping overhead.

Performance modeling: For performance modeling, SLAQ [65] and Optimus [45] predicts runtime performance and ML job convergence in an online manner. Similar to Harmony, Optimus captures high-level computation and communication patterns for prediction, but its model is completely different from Harmony because of the isolated resource environment (e.g., container) without resource sharing nor contention between the jobs. Gandiva [58] lacks a clear performance model of co-located (packed) jobs, as the interference makes the performance unpredictable. Morpheus [35] and Zhang et al. [66] use historical information of repetitive jobs for modeling. 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.

MonoTask [44] simulates executions of Spark jobs using the MonoTask abstraction, which is similar to subtask abstraction of Harmony. But their main goal is to predict and find a way to improve the performance and the expected value of improvement (e.g., 10% shorter JCT by replacing HDDs to SSDs or by changing system

configuration). Also, MonoTask does not cover multi-job situations, and is designed for Spark, which does not support PS-style communications. On the other hand, Harmony models the performance of multiple co-located PS jobs and finally improves the performance with several multi-job optimized techniques.

4.7 Summary

Harmony is a scheduling framework optimized for multiple PS ML jobs to improve cluster resource utilization. Harmony co-locates ML jobs and coordinates them to share resources effectively by minimizing contention of co-located jobs with subtask execution model, where each subtask uses specific types of resources intensively. 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 memory pressure incurred by the increased number of co-located ML jobs with dynamic data reloading. Our experiments 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`.

Chapter 5

Conclusion

5.1 Summary

We have presented two different solutions for improving the performance of distributed machine learning in a cluster environment. Both solutions build an analytical performance model and dynamically reconfigure the system to maximize the performance. Specifically, they find the best system configuration or scheduling decision based on the performance model with runtime metrics and dynamically applies it to the system with minimal overhead. The difference is that Cruise focuses on resource configuration and workload partitioning in a single-job, rather Harmony schedules resources and tasks for multiple jobs in a shared cluster. Evaluations show that these two works successfully enable automatic resource-efficient execution of PS ML systems, freeing users from manual settings.

5.2 Future Work

5.2.1 Other Communication Architecture Support

In this dissertation, we have focused on PS architecture, which is one of the popular architectures of distributed ML training. However, our approaches in Cruise and Har-

mony are not fundamentally limited to PS architecture and we can extend our work to other communication architectures such as All-Reduce. Both PS and All-Reduce architectures are widely used by many recent distributed ML frameworks, because both architectures have their own strengths [37].

5.2.2 Deep Learning & GPU Resource Support

Our works have focused on classical ML applications, and does not cover DL applications, which are widely used especially for language modeling and image recognition workloads. Though DL is one of the categories of ML, extending our work to DL requires non-trivial works due to its special characteristics. First, DL models consist of multiple layers and repeat forward/backward passes. We need to analyze such execution of DL training and build a performance model, which is more complicated than classical ML models. Second, DL workloads are typically trained with accelerators like GPU. Especially Harmony needs a mechanism for sharing GPU resources between multiple jobs in a fine-grained manner.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *WSDM*, 2012.

[3] Y. Bao, Y. Peng, C. Wu, and Z. Li. Online job scheduling in distributed machine learning clusters. *arXiv preprint arXiv:1801.00936*, 2018.

[4] J. Bennett and S. Lanning. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35, 2007.

[5] S. Boag, P. Dube, B. Herta, W. Hummer, V. Ishakian, K. JAYARAM, M. Kalantar, V. Muthusamy, P. NAG-PURKAR, and F. Rosenberg. Scalable multi-framework multi-tenant lifecycle management of deep learning training jobs. In *Workshop on ML Systems, NIPS*, 2017.

[6] M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve, et al. SystemML: Declarative machine learning on spark. *Proceedings of the VLDB Endowment*, 9(13):1425–1436, 2016.

[7] Carnegie Mellon University. Petuum Bösen, 2019. `https://github.com/sailing-pmls/bosen`.

[8] Carnegie Mellon University. Petuum Bösen, 2019. `https://github.com/sailing-pmls/bosen`.

[9] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz. Revisiting distributed synchronous sgd. In *International Conference on Learning Representations Workshop Track*, 2016.

[10] J. Chen, J. Zhu, J. Lu, and S. Liu. Scalable training of hierarchical topic models. volume 11, pages 826–839, 2018.

[11] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, 2014.

[12] B.-G. Chun, T. Condie, Y. Chen, B. Cho, A. Chung, C. Curino, C. Douglas, M. Interlandi, B. Jeon, J. S. Jeong, et al. Apache REEF: Retainable evaluator execution framework. *ACM Transactions on Computer Systems (TOCS)*, 35(2):5, 2017.

[13] B.-G. Chun, T. Condie, Y. Chen, B. Cho, A. Chung, C. Curino, C. Douglas, M. Interlandi, B. Jeon, J. S. Jeong, et al. Apache REEF: Retainable evaluator execution framework. *ACM Transactions on Computer Systems (TOCS)*, 35(2):5, 2017.

[14] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. P. Xing. Solving the straggler problem with bounded staleness. In *HotOS*, 2013.

[15] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting bounded staleness to speed up big data analytics. In *USENIX ATC*, 2014.

[16] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *EuroSys*, page 4. ACM, 2016.

[17] C. Curino, S. Krishnan, K. Karanasos, S. Rao, G. M. Fumarola, B. Huang, K. Chaliparambil, A. Suresh, Y. Chen, S. Heddaya, et al. Hydra: a federated resource manager for data-center scale analytics. In *NSDI*, 2019.

[18] J. Dai, Y. Wang, X. Qiu, D. Ding, Y. Zhang, Y. Wang, X. Jia, C. Zhang, Y. Wan, Z. Li, et al. BigDL: A distributed deep learning framework for big data. *arXiv preprint arXiv:1804.05839*, 2018.

[19] W. Dai, A. Kumar, J. Wei, Q. Ho, G. Gibson, and E. P. Xing. High-performance distributed ml at scale through parameter server consistency models. In *AAAI*, 2015.

[20] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *NIPS*, 2012.

[21] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS*, 2007.

[22] D. Dheeru and E. Karra Taniskidou. UCI machine learning repository, 2017.

[23] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald. Compressed linear algebra for large-scale machine learning. *The VLDB Journal—The International Journal on Very Large Data Bases*, 27(5):719–744, 2018.

[24] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, volume 11, pages 24–24, 2011.

[25] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo. Tiresias: A {GPU} cluster manager for distributed deep learning. In *NSDI*, pages 485–500, 2019.

[26] I. Gurobi Optimization. Gurobi optimizer, 2019.

[27] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Addressing the straggler problem for iterative convergent parallel ml. In *SoCC*, 2016.

[28] A. Harlap, A. Tumanov, A. Chung, G. R. Ganger, and P. B. Gibbons. Proteus: Agile ml elasticity through tiered reliability in dynamic resource markets. In *EuroSys*, 2017.

[29] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, volume 11, pages 261–272, 2011.

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

[31] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *NIPS*, 2013.

[32] B. Huang, M. Boehm, Y. Tian, B. Reinwald, S. Tatikonda, and F. R. Reiss. Resource elasticity for large-scale machine learning. In *SIGMOD*, 2015.

[33] M. Jeon, S. Venkataraman, J. Qian, A. Phanishayee, W. Xiao, and F. Yang. Multi-tenant gpu clusters for deep learning workloads: Analysis and implications. In *USENIX ATC*. USENIX Association, 2019.

[34] J. Jiang, B. Cui, C. Zhang, and L. Yu. Heterogeneity-aware distributed parameter servers. In *SIGMOD*, pages 463–478. ACM, 2017.

[35] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, Í. Goiri, S. Krishnan, J. Kulkarni, et al. Morpheus: Towards automated slos for enterprise clusters. In *OSDI*, pages 117–134, 2016.

[36] J. K. Kim, Q. Ho, S. Lee, X. Zheng, W. Dai, G. A. Gibson, and E. P. Xing. STRADS: a distributed framework for scheduled model parallel machine learning. In *EuroSys*, 2016.

[37] S. Kim, G.-I. Yu, H. Park, S. Cho, E. Jeong, H. Ha, S. Lee, J. S. Jeong, and B.-G. Chun. Parallax: Sparsity-aware data parallel training of deep neural networks. In *EuroSys*, page 43. ACM, 2019.

[38] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. MLbase: A distributed machine-learning system. In *CIDR*, volume 1, pages 2–1, 2013.

[39] H. Li, A. Kadav, E. Kruus, and C. Ungureanu. MALT: Distributed data-parallelism for existing ml applications. In *EuroSys*, 2015.

[40] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

[41] M. Li, D. G. Andersen, A. J. Smola, and K. Yu. Communication efficient distributed machine learning with the parameter server. In *NIPS*, pages 19–27, 2014.

[42] M. Li, L. Zhou, Z. Yang, A. Li, F. Xia, D. G. Andersen, and A. Smola. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop*, volume 6, page 2, 2013.

[43] T. Li, J. Zhong, J. Liu, W. Wu, and C. Zhang. Ease. ml: Towards multi-tenant resource sharing for machine learning workloads. volume 11, pages 607–620. VLDB Endowment, 2018.

[44] K. Ousterhout, C. Canel, S. Ratnasamy, and S. Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *SOSP*, 2017.

[45] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *EuroSys*, 2018.

[46] A. Qiao, A. Aghayev, W. Yu, H. Chen, Q. Ho, G. A. Gibson, and E. P. Xing. Litz: Elastic framework for high-performance distributed machine learning. In *USENIX ATC*, 2018.

[47] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. In *Proceedings of the VLDB Endowment (PVLDB)*, volume 3, pages 703–710, Sept. 2010.

[48] E. R. Sparks, A. Talwalkar, D. Haas, M. J. Franklin, M. I. Jordan, and T. Kraska. Automating model search for large scale machine learning. In *SoCC*, 2015.

[49] S. J. Subramanya, H. V. Simhadri, S. Garg, A. Kag, and V. Balasubramanian. Blas-on-flash: An efficient alternative for large scale ML training and inference? In *NSDI*, 2019.

[50] P. Sun, Y. Wen, N. B. D. Ta, and S. Yan. Towards distributed machine learning in shared clusters: A dynamically-partitioned approach. In *Smart Computing (SMARTCOMP), 2017 IEEE International Conference on*, pages 1–6. IEEE, 2017.

[51] The Apache Software Foundation. Apache Hadoop, 2015. `http://hadoop.apache.org`.

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

[53] P. Watcharapichat, V. L. Morales, R. C. Fernandez, and P. Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. In *SoCC*, 2016.

[54] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *SoCC*, 2015.

[55] M. Weimer, Y. Chen, B.-G. Chun, T. Condie, C. Curino, C. Douglas, Y. Lee, T. Majestro, D. Malkhi, S. Matusevych, B. Myers, S. Narayanamurthy, R. Ramakrishnan, S. Rao, R. Sears, B. Sezgin, and J. Wang. REEF: Retainable Evaluator Execution Framework. In *SIGMOD*, 2015.

[56] J. Wilkes and C. Reiss. Google cluster workload traces, 2015. `https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md`.

[57] T. Wojciech. Kubernetes scalability, 2017. `https://kubernetes.io/2017/03/scalability-updates-in-kubernetes-1.6.html`.

[58] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, et al. Gandiva: introspective cluster scheduling for deep learning. In *OSDI*, 2018.

[59] 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. In *SIGKDD*, 2015.

[60] F. Yan, O. Ruwase, Y. He, and T. Chilimbi. Performance modeling and scalability optimization of distributed deep learning systems. In *SIGKDD*, 2015.

[61] L. Yao, D. Mimno, and A. McCallum. Efficient methods for topic model inference on streaming document collections. In *SIGKDD*, 2009.

[62] L. Yut, C. Zhang, Y. Shao, and B. Cui. Lda*: a robust and large-scale topic modeling system. volume 10, pages 1406–1417, 2017.

[63] M. Zaharia. Job scheduling with the fair and capacity schedulers. *Hadoop Summit*, 9, 2009.

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

[65] H. Zhang, L. Stafman, A. Or, and M. J. Freedman. SLAQ: quality-driven scheduling for distributed machine learning. In *SoCC*, 2017.

[66] Y. Zhang, G. Prekas, G. M. Fumarola, M. Fontoura, Í. Goiri, and R. Bianchini. History-based harvesting of spare cycles and storage in large-scale datacenters. In *OSDI*, pages 755–770, 2016.

요약

기계 학습 시스템은 데이터에 숨겨진 의미를 뽑아내기 위해 널리 사용되고 있다. 데이터셋의 크기와 모델의 복잡도가 어느때보다 커짐에 따라 효율적인 분산 기계 학습 시스템을 위한 많은 노력들이 이루어지고 있다. 파라미터 서버 방식은 거대한 스케일의 데이터와 복잡한 모델을 지원하기 위한 유명한 방법들 중 하나이다. 이 방식에서, 학습 작업은 분산 워커와 서버들로 구성되고, 워커들은 할당된 입력 데이터로부터 반복적으로 그레디언트를 계산하여 서버들에 보관된 글로벌 모델 파라미터들을 업데이트한다.

파라미터 서버 시스템의 성능을 향상시키기 위해, 이 논문에서는 자동적으로 자원 효율성과 시스템 성능을 최적화하는 두가지의 해법을 제안한다. 첫번째 해법은, 파라미터 시스템에서 분산 기계 학습을 수행시에 자원 설정 및 워크로드 분배를 자동화하는 것이다. 최고의 설정을 찾기 위해 우리는 온라인 메트릭을 사용하는 비용 모델을 기반으로 하는 Optimizer를 만들었다. Optimizer의 결정을 효율적으로 적용하기 위해, 우리는 런타임을 동적 재설정을 최소의 오버헤드로 백그라운드에서 수행하도록 디자인했다.

두번째 해법은 공유 클러스터 상황에서 여러 개의 기계 학습 작업의 세부 작업과 자원의 스케쥴링을 최적화한 것이다. 구체적으로, 우리는 세부 작업들을 세밀한 단위로 수행함으로써 자원 경쟁을 억제하고, 서로를 보완하는 자원 사용 패턴을 보이는 작업들을 같은 자원에 함께 위치시켜 자원 활용율을 끌어올렸다. 함께 위치한 작업들의 메모리 압력을 경감시키기 위해 우리는 동적으로 데이터를 디스크로 내렸다가 다시 메모리로 읽어오는 기능을 지원함과 동시에, 디스크와 메모리간의 데이터 비율을 상황에 맞게 시스템이 자동으로 맞추도록 하였다.

위의 해법들을 실체화하기 위해, 실제 동작하는 시스템을 만들었다. 두가지의 해법을 하나의 시스템에 구현함으로써, 동적으로 작업을 머신 간에 옮기고 자원을 재할당할 수 있는 런타임을 공유한다. 해당 솔루션들의 효과를 보여주기 위해, 이 시스템을 많이 사용되는 기계 학습 어플리케이션으로 실험하였고 기존 시스템들 대비

뛰어난 성능 향상을 보여주었다.

주요어: 기계학습, 분산 트레이닝, 파라미터 서버, 비용 기반 성능 모델링, 동적 최적화, 작업/자원 스케쥴링
학번: 2013-23132