Scaling Distributed Machine Learning with System and Algorithm Co-design

Mu Li

February 2017

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
David Andersen, co-chair
Jeffrey Dean (Google)
Barnabas Poczos
Ruslan Salakhutdinov
Alexander Smola, co-chair

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Copyright © 2017 Mu Li

Keywords: Large Scale Machine Learning, Distributed System, Parameter Server, Distributed Optimization Method

Dedicated to my lovely wife, QQ.

Abstract

Due to the rapid growth of data and the ever increasing model complexity, which often manifests itself in the large number of model parameters, today, many important machine learning problems cannot be efficiently solved by a single machine. Distributed optimization and inference is becoming more and more inevitable for solving large scale machine learning problems in both academia and industry. However, obtaining an efficient distributed implementation of an algorithm, is far from trivial. Both intensive computational workloads and the volume of data communication demand careful design of distributed computation systems and distributed machine learning algorithms. In this thesis, we focus on the co-design of distributed computing systems and distributed optimization algorithms that are specialized for large machine learning problems.

In the first part, we propose two distributed computing frameworks: Parameter Server, a distributed machine learning framework that features efficient data communication between the machines; MXNet, a multi-language library that aims to simplify the development of deep neural network algorithms. We have witnessed the wide adoption of the two proposed systems in the past two years. They have enabled and will continue to enable more people to harness the power of distributed computing to design efficient large-scale machine learning applications.

In the second part, we examine a number of distributed optimization problems in machine learning, leveraging the two computing platforms. We present new methods to accelerate the training process, such as data partitioning with better locality properties, communication friendly optimization methods, and more compact statistical models. We implement the new algorithms on the two systems and test on large scale real data sets. We successfully demonstrate that careful co-design of computing systems and learning algorithms can greatly accelerate large scale distributed machine learning.

Acknowledgments

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past decade, machine learning (ML) prevailed in both industry and in academia. ML has a wide range of applications in automated document analysis, computer vision, natural language processing, voice recognition and computational advertising. For all the technological advances in the field of ML, there are two prevalent trends: the size of training data is getting larger ("bigger and bigger data"), and the statistical models are becoming more complicated ("deeper and deeper models"). In this thesis, we aim to address these two issues of large scale machine learning by exploring the co-design of distributed computing systems and distributed learning algorithms.

Both "big data" and "deep learning" significantly increase the computational cost of ML applications. For example, companies often need to train ML models from business data of terabytes [27]. A state-of-the-art image classifier usually employs hundreds of layers in a convolutional neural network model [64], in which processing a single image requires billions of float-point operations. At such a large scale, although the computing power of modern hardware grows exponentially, no single machine can finish the training tasks within a time frame that meets the industrial demands.

Distributed computing is a common approach to tackle the problem of large scale machine learning. The basic idea of distributed computing is to partition the computational workload and assign different parts to different computing machines. Then the machines coordinate to complete the task. Recently, thanks to the increasingly convenient access to public cloud services, such as Amazon AWS [8], Google Cloud [61] and Microsoft Azure [100], using distributed computing to accelerate large scale machine learning has led to a surge of research interest in both academia and industry.

However, both the design of an efficient distributed computing system and efficient implementation of ML algorithms in the system are highly non-trivial. A major challenge comes from the communication cost in the distributed computing environment. In particular, both the iterative nature of many machine learning algorithms and the sheer size of the models and the training data require a large amount of communication between different machines in the training process. However, today, even in the leading industrial data centers, both the network bandwidth and the communication latency between machines is at least 10 times worse than communicating within a single machine. The communication overhead is indeed the major bottleneck that prevents us from applying distributed computing to solve large scale machine learning problems.

There are also other challenges in using distributed computing, such as enhancing fault tolerance so that the computation is not interrupted when one or more machines break down, which all need to be carefully addressed to make it practical for large scale machine learning.

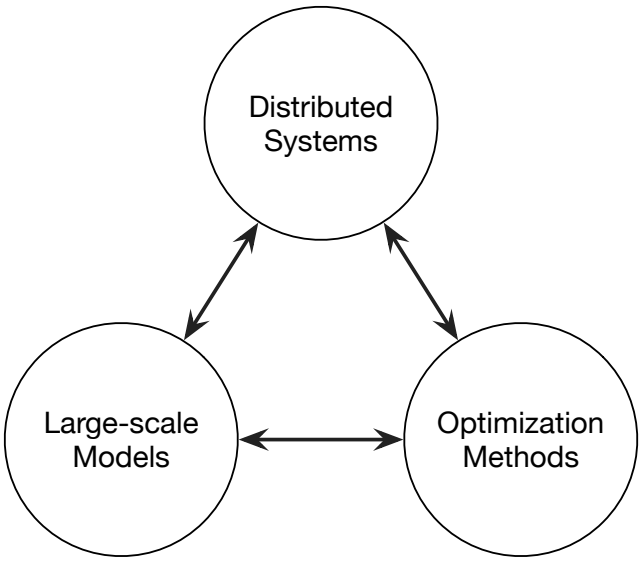


Figure 1.1: Three aspects towards distributed large scale machine learning.

1.1 Background

In this thesis, we tackle the problem of large scale machine learning from three aspects: distributed computing systems, large scale models and optimization methods. For a wide class of machine learning applications, we design new distributed computing frameworks, study new machine learning models, and propose new optimization methods to shows that distributed machine learning can be made simple, fast, and scalable. As illustrated in Figure 1.1, these three aspects are closely connected. Next, we give a briefly overview of each aspect and highlight the corresponding challenges.

1.1.1 Large Scale Models

The realm of machine learning is mainly divided into supervised and unsupervised learning. In supervised learning, the training data consists of pairs of input and output values, and the learning goal is to infer a mapping from input to output, which can be used for predicting the output for new input instances. An example for supervised learning is image classification, where one example of the training data is a pair of the input image and the corresponding output label, indicating the name of the object contained in the image. In unsupervised learning, the training data contains only the input values, and the learning goal is to find "interesting patterns" in the training data. One example of unsupervised learning is to cluster input data points in a way such

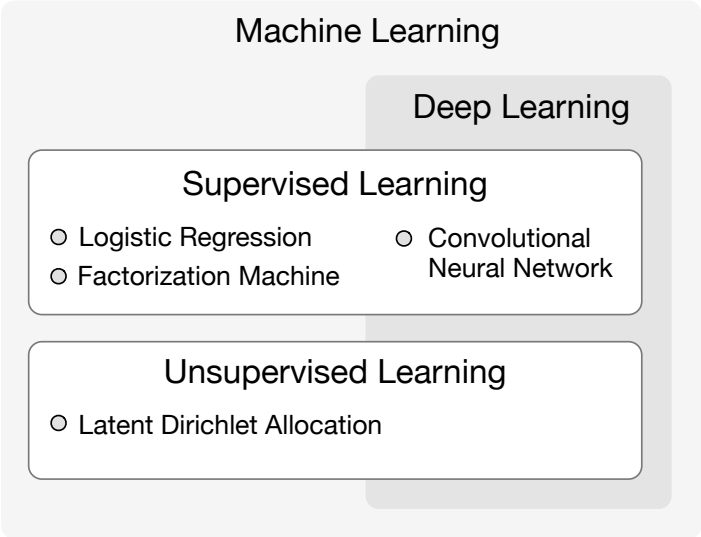


Figure 1.2: Machine learning algorithms studied in this thesis.

that the data points in the same cluster are more similar to each other compared to points across different clusters.

In this thesis, we consider several representative ML algorithms in each category as motivating examples for our system and algorithm co-design. These applications range from simple linear models to complex neural network models with hundreds of layers. We list and classify these algorithms in Figure 1.2. One example of the applications is logistic regression for click-through rate estimation, which aims to learn a map from an ad impression to the probability that this ad is clicked by an end user. Another example is to learn a latent Dirichlet allocation that identifies the topics that a user is in from the user browser history.

A recent breakthrough in both supervised and unsupervised machine learning is called *deep learning*. In conventional statistical models, examples of the training data are often represented as multiple dimensional vectors, and representation is obtained from the raw data by *feature extraction*. For instance, we can use n-grams to encode word documents, or use scale-invariant feature transformations [92] to describe local features in images. Instead of using hand-crafted features, deep learning serves to automatically "learn" a feature representation from the raw data, by training a multi-layer neural network.

Today, a common challenge for different machine learning problems is the rapidly increasing size of training data and the growing model complexity. For instance, a large Internet company wants to use one year's ad impression log [74] to train an ad-click predictor. The training data consists of trillions of examples, each of which is typically represented by a high-dimensional feature vector [27]. Figure 1.3 shows the size of training data for the problem of ad-click estimation in a large Internet company. Note that the size of the data almost doubled every year form 2010 to 2014. A lot of widely used machine learning algorithms process the training data in an iterative way. When the training data is at such a large scale, the amount of computing resources required is enormous. Moreover, billions of new ad impressions are generated everyday.

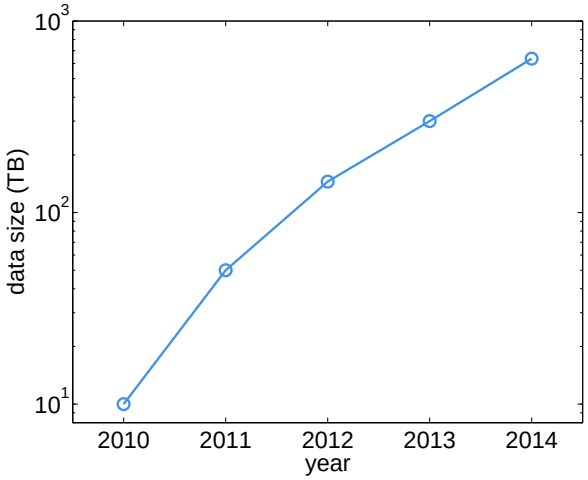


Figure 1.3: The size of training data for ad click estimation in a large Internet company from year 2010 to 2014.

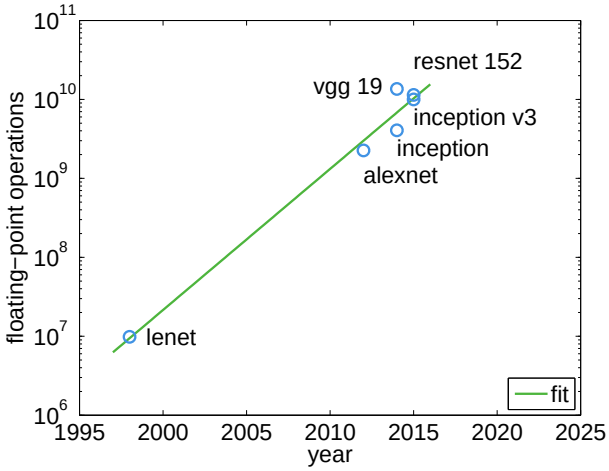


Figure 1.4: The number of floating-point operations required for processing a single image using LeNet and several recent ImageNet challenge winners.

It improves the ad-click prediction accuracy if the real-time data is incorporated in the training process, but real-time large scale learning imposes even greater challenges [99].

Meanwhile, large size of training data enable and encourage the engineers to use more complex machine learning models to discover finer structures in the data. Take deep learning as an example, the model size in terms of the depth of the neural networks has been consistently increasing since the 1980s. In 1989, LeNet, one widely used convolutional neural network, only had 5 convolutional layers; while all the recent ImageNet challenge winners [64, 134] employed hundreds of convolutional layers. More complex models are often associated with higher computational cost. Figure 1.4 shows that the number of floating-point operations required in order to process a single example increased from 10 million to over 10 billion over the 20 years. Besides the computational cost, deeper neural networks also bring in more complex computational patterns—even just evaluating a single example involves hundreds of tensor operations.

Therefore, rapidly increasing amount of training data and more complex machine learning models necessitate new solutions of the design of both computational system and machine learning algorithm.

1.1.2 Distributed Computing

As shown in Figure 1.5, a distributed computing system consists of multiple nodes with computing power, and the nodes are connected through a communication network. Examples of distributed computing system range from distributed scientific computing on supercomputers to distributed autonomous sensors for monitoring physical conditions. In this thesis, we focus on *cluster computing*, where actual computers are connected via local communication networks. Examples of cluster computing include campus cluster machines, as well as public cloud service

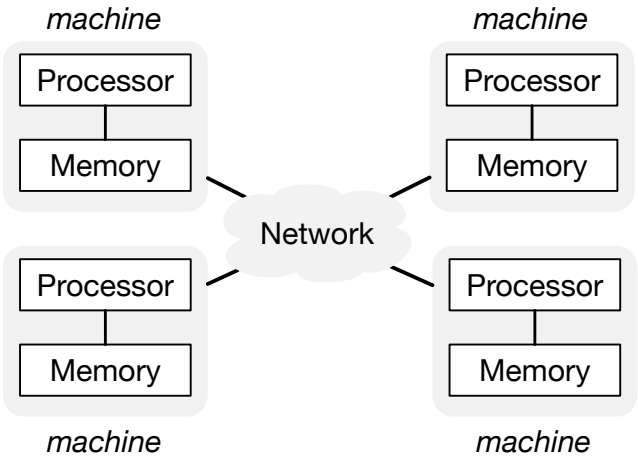


Figure 1.5: A distributed computing system with distributed memory

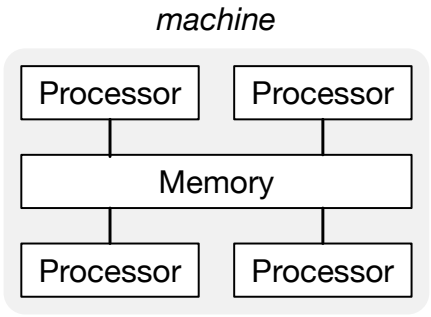


Figure 1.6: A distributed computing system with shared memory

	parallel job	distributed job
# of CPUs	4	1000s
# of GPUs	8	100s
latency	100 ns	0.1 ms
bandwidth	400 Gbit/sec	10 Gbit/sec

Table 1.1: Statistics for typical parallel and distributed jobs.

such as Amazon AWS [8], Google Cloud [61] and Microsoft Azure [100]. In cluster computing, the structure of the system and the network topology are known in advance.

Distributed computing is different from a commonly used technology called *shared-memory parallel computing*, which assumes that the processors are located within a small distance to each other, so that they have access to the same piece of shared memory. In a distributed computing system, each computing machine has its own private memory, which cannot be directly accessed by another machine. The difference between distributed-memory and shared-memory can be seen from Figure 1.5 and Figure 1.6. Table 1.1 shows that compared to shared-memory parallel computing, distributed computing systems usually have many more processors and can handle more computational jobs simultaneously.

For distributed computing, information exchange between machines is conducted over the communication network, which has limited bandwidth. Indeed, communication is one of the scarcest resources in a distributed computing system. Moreover, a machine may fail at any time, and a running job can be preempted. Such unreliability of a system becomes worse when the number of machines and the size of workload increase. These properties of distributed systems impose great challenges on developing efficient and reliable machine learning applications using distributed computing.

$\approx$ #machine $\times$ time	# of jobs	failure rate
100 hours	13,187	7.8%
$1,000$ hours	1,366	13.7%
$10,000$ hours	77	24.7%

Table 1.2: Failure rate for machine learning jobs in a data center over a three month period.

There three desired properties that are most important for a distributed computing system: efficiency, fault tolerance, and easy-to-use.

Efficiency An efficient distributed computing system incurs a small communication cost and takes full advantage of the available computing power in the system.

Apart from the actual computational cost that is shared among multiple machines, distributed computing incurs additional cost of communication overhead and machine synchronization. Compared to accessing the memory to retrieve information within a single machine, both the communication latency and the communication bandwidth in a distributed computing system are significantly worse. For example, the latency for main memory accessing is in the order of magnitude of 100ns, while it is in the order of 0.1 ms to 1ms between machines in a data center. The memory bandwidth in a personal computer is around 400 Gbit/sec, while a typical network bandwidth provided by Amazon AWS is only 10 Gbit/sec. Moreover, such limited communication bandwidth is shared among all the computing machines and among all the running tasks.

In addition, the computing machines in the system may be very different, or they may run heterogeneous computational tasks. Thus, some machines may finish their tasks faster or slower. In this case, if machine synchronization is required for implementing an algorithm, the computational power of all the machines may not be fully used at any time when there are stragglers in the system.

Good system design reduces the communication overhead and the effects of machine synchronization.

Fault Tolerance In a distributed computing system, a single machine may fail at any time, and a running job can be preempted due to such machine breakdown. Moreover, the number of failures increases with the number of machines in the systems and increases with the size of the computational tasks. We collect the job logs from a computing cluster that runs machine learning tasks for production in a large Internet company, over a three month period. Here, task failures are mostly due to being preempted or machine breakdown. Table 1.2 shows the statistics of failure rate for different tasks at various scale. Observer that the failure rate can be as high as 25% for large scale problems that need over 10 thousand machine hours of computation.

An important feature of a good distributed computing system is the fault tolerance, namely its robustness to such failures.

Easy to use It is also desired that the programming interface of a distributed system strikes a balance between simplicity and flexibility. On one hand, the interface should hide as much as possible implementation details from the application developers. The implementation details

of using multiple machines include task partition and allocation, data communication, machine synchronization and fault tolerance mechanism. On the other hand, the interface should be flexible enough so that the developers can conveniently implement a wide range of algorithms using the system.

There exist different approaches for application program interface (API) design for distributed computing system. For example, MapReduce [41] is one of the most widely used frameworks. However, the synchronization forced at the end of each map and reduce cycle potentially limits the performance of iterative machine learning algorithms. Another example is Message Passing Interface (MPI). It provides flexible routines for high performance data communication, yet exposes the developers to way too many implementation details such that reliable programming can be challenging in this system.

1.1.3 Optimization Methods

In machine learning, training a statistical model can often be formulated into an optimization problem [143] in the following form:

$$\underset{w\in\Omega}{\text{minimize}}\ f(w) = \frac{1}{n}\sum_{i=1}^{n} f_i(w), \tag{1.1}$$

where w denotes the model parameter and Ω denotes the parameter space. The function f_i is the objective function evaluated with i-th example, describing how well the model w fits the particular example in the training data. Consider the standard linear regression. The i-th example in the training data consists of a p-dimensional vector x_i and an output value y_i. The goal of model training is to find a p-dimensional vector model w, so that given a new example we can approximately predict the output value y with the value $\langle w, x\rangle$. In order to achieve a small prediction error, one possible objective function is the Euclid distance between $\langle w, x_i\rangle$ and y_i, namely $f_i(w) = \| \langle w, x_i\rangle - y_i\|_2^2$.

For general objective function f_i, there is usually no explicit solution to the optimization problem. A common way to get a numerical solution is the iterative gradient method, which refines the model parameter w through multiple iterations of gradient flow operations. The basic idea is to start with an initial point $w_0 \in \Omega$ at $t = 0$, and then update the model as below:

$$w_{t+1} = \underset{\Omega}{\text{proj}}\left[w_t - H_t \sum_{i\in I_t} \partial f_i(w_t)\right] \quad \text{for } I_t \subseteq \{1,\ldots,n\}, \tag{1.2}$$

where ∂f_i is the partial gradient of f_i with respect to the model parameter w, H_t is a p-by-p scaling matrix or scalar, and the set I_t is a subset of example indices which are processed at iteration t. Different choices of H_t and I_t lead to different optimization methods. For instance, the standard gradient descent method can be obtained by setting $I_t = \{1,\ldots,n\}$ and setting H_t to be a constant scalar η. The iterations terminate if a stopping criteria is reached.

The bottleneck of iterative gradient methods is often the cost of calculating the gradients $\partial f_i(w)$ in each iteration. It is possible to partition and share this computational workload among

Algorithm 1 Distributed gradient-based optimization

Initialize w_0 at every machine
for $t = 0, \ldots$ **do**
  Partition $I_t = \bigcup_{k=1}^{m} I_{t_k}$
  for $k = 1, \ldots, m$ **do** in parallel
    Compute $g_t^{(k)} \leftarrow \sum_{i \in I_{t_k}} \partial f_i(w_t)$ on machine k
  end for
  Aggregate $g_t \leftarrow \sum_{k=1}^{m} g_t^{(k)}$ on machine 0
  Update $w_{t+1} \leftarrow w_t - H_t^{-1} g_t$ on machine 0
  Broadcast w_{t+1} from machine 0 to all machine
end for

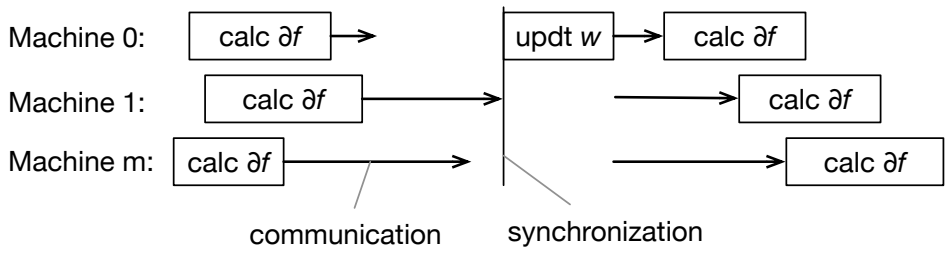


Figure 1.7: The communication and synchronization overhead of Algorithm 1.

multiple machines. Algorithm 1 sketches the approach called data parallelism. In each iteration, the training data is partitioned and assigned to different machines, and each machine only calculates the gradients of the objective function evaluated on the assigned training data.

One important measure of the performance of an iterative optimization method is its convergence rate, namely the amount of computing time required in order to achieve a desirable accuracy. There is a rich body of research results on accelerating the vanilla gradient descent method. For example, stochastic gradient descent (SGD) [117] only samples a small subset of examples in the training data to form the index set I_t, so that each iteration takes much less time. Given the same amount of total run time, this approach can afford more number of iterations to updates the model parameter, and hopefully obtains a better value of the objective function.

When data parallelism is used, another measure of the performance of the optimization method is the communication cost, which includes the amount of communication required between the machines, and the amount of time that the machines stay idle due to communication latency or for the purpose of system synchronization. For example, Figure 1.7 shows a possible distributed implementation of Algorithm 1. The problem with this straightforward implementation is that if the data is partitioned unevenly or one machine is significantly slower than other machines, a large communication cost is incurred due to machine synchronization.

A good distributed optimization method should achieve fast convergence with small communication cost.

1.2 Thesis Statement

This thesis seeks to address the multifaceted challenges arising in distributed computing, optimization methods, and large scale models to make large scale distributed machine learning more accessible. In particular, this thesis provides evidence to support the following statement:

> **Thesis Statement:** *With appropriate computational frameworks and algorithm design, distributed machine learning can be made simple, fast, and scalable, both in theory and in practice.*

We believe that the computational frameworks and the algorithmic ideas developed in this thesis will enable more people to take advantage of the power of distributed computing to develop efficient machine learning applications to solve large scale problems. All the codes developed when completing this thesis are made publicly available at `https://github.com/dmlc/` under Apache 2.0 license.

1.3 Thesis Contributions

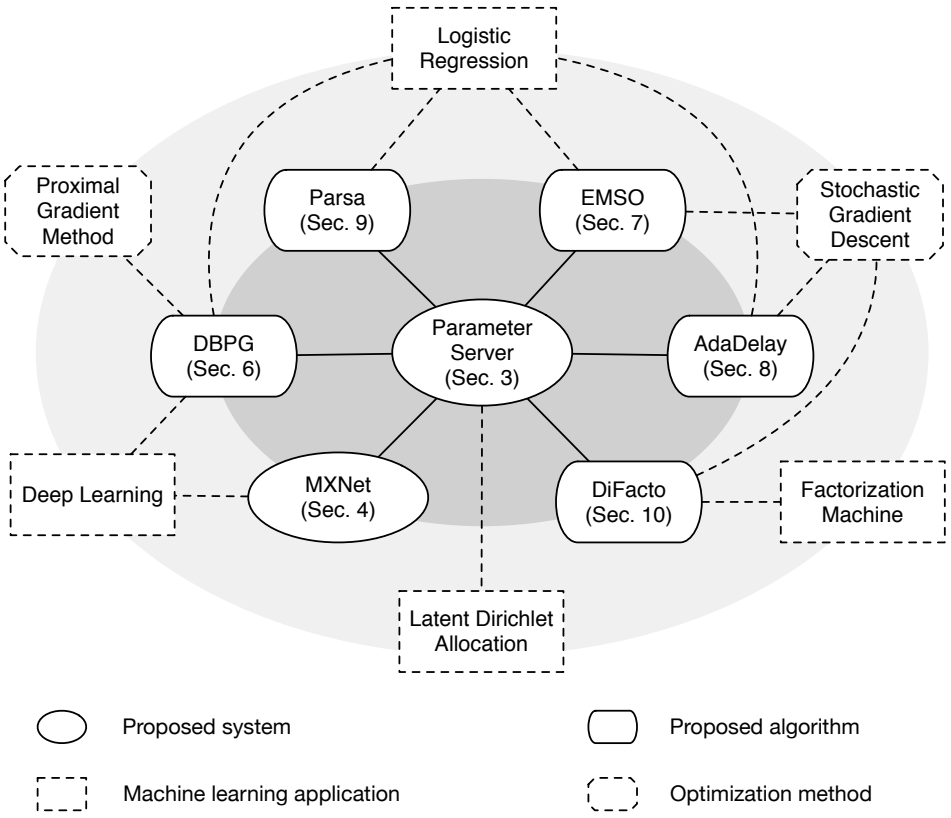


Figure 1.8: Connections between the proposed systems and the algorithms

Our major contributions in these three areas are summarized below:

Distributed systems We design new distributed computing systems that are optimized for machine learning tasks. Our systems provide a high-level system abstraction so that the developers can focus on designing machine learning algorithms without worrying about the implementation details. We demonstrate that our systems are highly efficient and scalable.

Optimization methods We propose system-friendly optimization methods that can be easily parallelized and implemented in a distributed computing system. We show that the methods achieve fast convergence with reduced communication overhead.

Large scale models Given a fixed amount of training data, a complex model may create the problem of overfitting, which is often resolved by model parameter regularization. In a distributed computing environment, a complex model also introduces a large amount of communication overhead between machines. We explore the approach of regularization to design large scale models with sparse parameters, which are further exploited to reduce the communication overhead as well as computational cost.

Next, we briefly describe the tentative structure of the thesis. The thesis is mainly divided into two parts: distributed computing system (Section 2 to 4), distributed optimization methods (Section 5 to 10). Figure 1.8 visualizes the connections between different sections.

Part I In the first part of the thesis, we introduce two computing frameworks designed for large scale distributed machine learning: a new generation of Parameter Server and MXNet. The key features of these two systems are summarized below.

Parameter Server (PS) PS is a general purpose distributed machine learning framework. Compared to existing frameworks, it has the following prominent features:

- Asynchronous communication, which is optimized for machine learning tasks to reduce network communication overhead.
- Flexible consistency model, which further lowers the synchronization cost and communication latency.
- Elastic scalability, which enables adding new machines to the system without restarting the running framework.
- Continuous fault tolerance, which prevents non-catastrophic machine failures to interrupt the overall computation process.
- An efficient implementation of vector clocks, which ensures well-defined behavior after network failures.

This is joint work with David Andersen, Alex Smola, and Junwoo Park from CMU, together with Amr Ahmed, Vanja Josifovski, James Long, Eugene Shekita and Bor-Yiing Su from Google. Part of the work has been published in OSDI'14 [83].

MXNet MXNet is a multi-language library aiming to simplify the algorithm development for large scale deep neural networks. It outperforms many existing platforms by exploiting the following features:

- A mixed interface with imperative programming and symbolic programming to achieve both flexibility and efficiency.
- A compiler-like back-end system that efficiently optimizes workloads.
- A more general asynchronous execution engine that extends Parameter Server's asynchronous communication model to alleviate the data dependencies in deep neural

networks.
- More flexible ways for using heterogeneous computing.

This is joint work with a large number of collaborators from difference university and companies, including Tianqi Chen (U. Washington), Yutian Li (Standford), Min Li (NUS), Naiyan Wang (TuSimple), Minjie Wang (NYC), Tianjun Xiao (Microsoft), Bing Xu (Apple), Chiyuan Zhang (MIT), and Zheng Zhang (NYU Shanghai). Part of the work has been published in the learning system workshop on NIPS'16 [31].

Part II In the second part of the thesis, we present new distributed optimization algorithms for several machine learning problems. We implement the algorithms on either PS or MXNet and demonstrate that careful co-design of computing systems and optimization algorithms can greatly accelerate large scale distributed machine learning.

We first study the problem of speeding up coordinate descent and stochastic gradient descent (SGD), which are widely used optimization methods in distributed computing environments.

DBPG To speed up coordinate descent, we propose a new algorithm named **DBPG** based on the proximal gradient method to solve non-convex and non-smooth problems. It updates parameters in the blockwise style and allows delays between blocks to reduce the synchronization cost. Theoretical analysis shows that the algorithm converges under weak assumptions.

This is joint work with David Andersen, Alexander Smola, together with Kai Yu from Baidu. The results were published in NIPS'14 [84]

EMSO To speed up SGD in parallel computing, minibatch training has been used to reduce the communication cost, at the cost of a slower convergence rate. We propose a new minibatch training algorithm called **EMSO**. Instead of just running gradient descent with each minibatch, for each minibatch the algorithm solves an optimization with a conservatively regularized objective function. We show that this more efficient use of minibatches speeds up the convergence while maintaining low communication cost.

This is joint work with Alexander Smola together with Tong Zhang and Yuqiang Chen from Baidu. The results were published in KDD'14 [85]

AdaDelay To speed up asynchronous SGD, we propose a new algorithm **AdaDelay**, which allows the parameter updates to be sensitive to the actual delays experienced, rather than to worst-case bounds on the maximum delay. We show that this delay sensitive update rule leads to larger stepsizes, that can help gain rapid initial convergence without having to wait too long for slower machines, while maintaining the same asymptotic complexity.

This is joint work with Suvrit Sra from MIT, together with Adams Yu and Alexander Smola from CMU, and the results were published in AISTATS'16 [127].

We then study the problem of using data partitioning to reduce the communication and synchronization cost.

Parsa We formulate data placement as a submodular load-balancing problem and we propose a parallel partition algorithm named **Parsa** to solve it approximately. We show that with high probability the objective function is at least $n/\log(n)$ of the that with the best partition. The runtime of the algorithm is in the order of $\mathcal{O}(k|E|)$, where k is the number of partitions and $|E|$ is the number of edges in the graph.

f	objective function
(x_i, y_i)	i-th data example
w	model parameter
g	gradient
$[v]_i$	the i-th coordinate of vector v
n	number of examples
p	number of parameters
m	number of machines
T	number of iterations

Table 1.3: Notations used in thesis.

This is joint work with David Andersen and Alexander Smola, and the results appeared in arXiv [86].

Finally we study a promising nonlinear model for recommendation and estimation—Factorization Machines. It has been shown that factorization machine can achieve much better performance compared to simple linear models. However this complex model increases the computation cost by at least an order of magnitude larger, which is the main barrier of its applications in practice.

DiFacto To make factorization machine scale to large amounts of data and large numbers of features, we propose a new algorithm **DiFacto**, which uses a refined Factorization Machine model with sparse memory adaptive constraints and frequency adaptive regularization. This is joint work with Ziqi Liu, Alexander Smola, and Yu-Xiang Wang. The results were published in WSDM'16 [87].

1.4 Notations, Datasets and Computing Systems

1.4.1 Notations

The notations used in this thesis are listed in Table 1.3.

1.4.2 Datasets

We used the following publicly available datasets in our experiments.

Binary text classification

RCV1[1] The documents come from Reuters Corpus Volume 1.

News20[2] The documents come from 20 newsgroups.

KDD04[3] The particle physics task in KDD Cup 2004. The goal is to classify two types of particles generated in high energy collider experiments.

[1] `http://www.daviddlewis.com/resources/testcollections/rcv1/`
[2] `http://qwone.com/~jason/20Newsgroups/`
[3] `http://osmot.cs.cornell.edu/kddcup/datasets.html`

name	examples	unique features	non-zero entries
RCV1	20 K	47 K	1 M
News20	20 K	1 M	9 M
KDD04	146 K	74	11 M
KDD14	8 M	20 M	305 M
URL	2.4 M	3.2 M	277 M
Criteo	1.9 B	360 M	58 B
CTRa	100 M	283 M	10 B
CTRb	170 B	65 B	17 T

Table 1.4: The datasets for binary text classification.

name	nodes	edges	type
LiveJournal	5M	69M	directed
Orkut	3M	113M	undirected

Table 1.5: The social network datasets.

KDD14[4] The first problem in KDD Cup 2010. The goal is to predict student performance on mathematical problems from logs of student interaction with Intelligent Tutoring Systems.

URL[5] The goal is to detect malicious URLs.

Criteo[6] The goal is to estimate the click-through rate of ads that come from Criteo.

CTR Similar to Criteo. The dataset comes a private Internet company. CTRa is sampled from a three-month period, while CTRb is sampled from a two-year period.

One-hot encoding is used for all datasets. More specifically, we form a dictionary using all the unique features. Then each example in the dataset is presented by a vector x whose length is equal to the size of the dictionary. The element $x_i = 1$ if the i-th word in the dictionary appears in this record, otherwise $x_i = 0$. Note that the size of the dictionary can be huge, yet the number of distinct features that appear in one example is usually very small. In other words, the datasets are often high dimensional, but extremely sparse.

We list the statistics of these datasets in Table 1.4.

Social Networks We have two datasets of the social network LiveJournal and Orkut, and they are obtained from `http://snap.stanford.edu/data/`. We list the statistics of these 2 datasets in Table 1.5.

Image Classification We use the ImageNet competition 2012[7] dataset, which contains 1.3M images from 1,000 classes.

[4] `https://pslcdatashop.web.cmu.edu/KDDCup/`
[5] `http://sysnet.ucsd.edu/projects/url/`
[6] `http://labs.criteo.com/downloads/download-terabyte-click-logs/`
[7] `http://image-net.org/challenges/LSVRC/2012/`

name	CPU	GPU	memory (GB)	network (Gbit/s)	# of machines
CompanyA	2×Intel Xeon series	-	192	10	1000
CompanyB	2×Intel Xeon series	-	128	≥ 10	5000
CampusA	2×Intel Xeon E5620	-	64	1	16
CampusB	4×AMD Opteron 6272	Tesla K20	128	40	36
CampusC	2×Intel Xeon E5-2680 v2	2×GTX 980	128	40	10
Desktop	Intel i7-2600	GTX 750 TI	32	1	1
EC2-g2.8x	2×Intel Xeon E5-2686 V4	8×Tesla K80	768	25	16
EC2-c4.8x	2×Intel Xeon E5-2666 V3	-	64	10	10

Table 1.6: Computing systems used in the experiments.

1.4.3 Computing systems

In Table 1.6, we list the specifications of the diverse clusters used in the experiments. They range from campus clusters to public and private cloud services.

Part I

System

Chapter 2

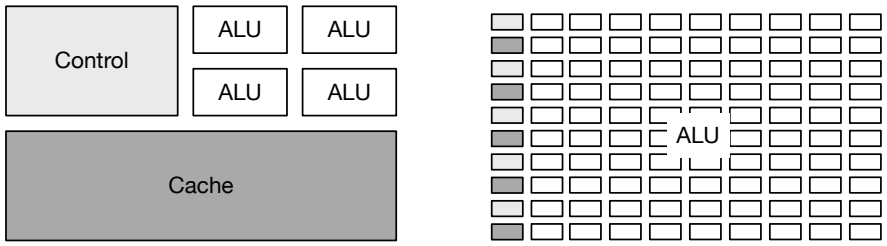
Preliminaries on Distributed Computing Systems

In this section we provide some background information of computing systems that are closely related to our proposed systems. In particular, we discuss issues of heterogeneous computing and data center. Heterogeneous computing is widely used to accelerate computation intensive workloads such as deep neural networks, and data center is where most distributed machine learning applications are running. All proposed systems and algorithms in this thesis are evaluated in these two computing environments.

2.1 Heterogeneous Computing

In a distributed computing system, for the sake of performance or energy efficiency, instead of using the same type of processors, it is often desirable to include different type of co-processors. Heterogeneous computing [121] refers to distributed computing systems that use more than one kind of processor or cores. Recently, it has been widely used in super-computing and cloud computing service. For example, all of the current top 3 supercomputers [3] are equipped with co-processors. Sunway TaihuLight uses on-broad slave cores, Tianhe-2 uses Intel Xeon Phi, and Titan uses NVIDIA K20x. Cloud computing providers including Amazon Web Service and Microsoft Azure are also providing instances installed with both GPUs and CPUs. In the rest of this section, we will be using system equipped with GPUs as an example of heterogeneous computing. Note that the techniques discussed here can be extended to other co-processors such as Xeon Phi.

The purpose of adding GPUs to the system is to leverage the computational power, as the computing power of GPUs far exceeds that of standard CPUs. For example, Pascal NVIDIA TITAN X provides 11 TFLOPS [1], while peak performances of high-end CPUs, such as Intel Xeon E5-2699 v4, is still less than 1 TFLOPS [2]. In terms of computer architecture, GPUs and CPUs are very dissimilar processors. As shown in Figure 2.1, in CPU, more than half of the chip is used for cache and control units, which makes CPU suitable for workloads with complex logic and irregular memory access patterns. However, the major chip area of GPUs is dedicated to arithmetic and logical operations. As a result, a GPU can often afford hundreds of



(a) CPU (b) GPU

Figure 2.1: The architecture of CPU and GPU.

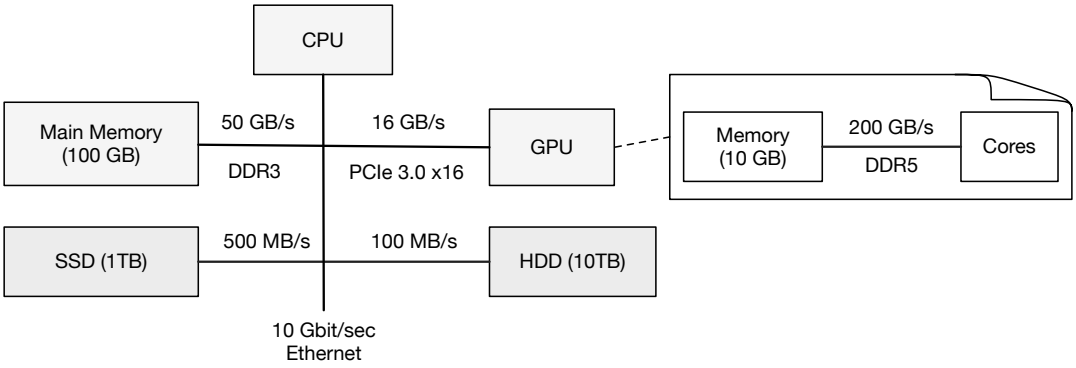


Figure 2.2: Typical capacity and bandwidth of system components.

computing threads, compared to just tens of threads for a CPU. Therefore GPUs well fit the more computation-extensive and highly paralleled workloads.

In heterogeneous computing, GPUs are usually connected to CPUs by PCI Express (PCIe). A single lane (x1 connection) of PCIe 3.0 contains two pairs of wires—one for sending and one for receiving, with a bandwidth of 0.985 GB/sec for each direction. When using a x16 configuration, a GPU can communicate with another device using a bandwidth of 15.75 GB/sec in each direction. Figure 2.2 shows the how a GPU and other system components are connected in a typical heterogeneous computing system. First note that accessing GPU via PCIe is more than 10 times faster than accessing the disk and the network, yet it is still significantly slower than accessing the main memory. Also, a GPU is often equipped with memory which can be accessed using high bandwidth by its core, and this bandwidth is typically about 4 times of that of the main memory.

Standard CPUs only provide a limited number of lanes for PCIe. For example, the Intel GPU generation 2600 v3 supports at most 40 lanes. In order to connect multiple GPUs to a CPU simultaneously and utilize the maximal bandwidth, we often use PCIe switches. Figure 2.3 shows the connection between 2 PCIe switches and 8 GPUs. Each PCIe switch connects 4 GPUs to 1 CPU and uses 16 lanes PCIe for each connection. The two CPUs are then connected by QPI, which has similar bandwidth as that of PCIe. Note that GPUs connected by the same PCIe switch enjoy full bidirectional bandwidth for peer-to-peer communication. However, the bottleneck of

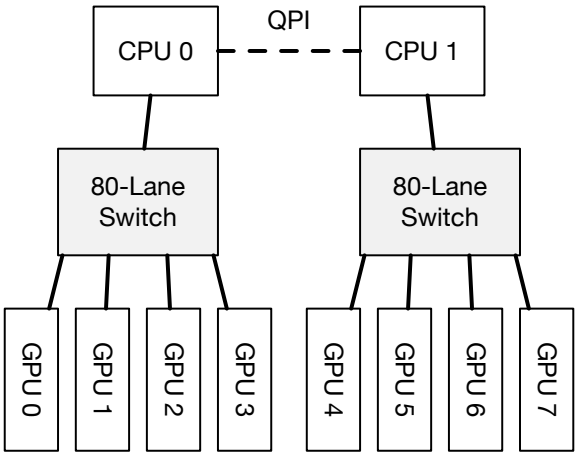


Figure 2.3: Connecting 8 GPUs to 2 CPUs via two PCIe switches. Each solid line represents 16 lanes.

communication between GPUs belonging to different switches is determined by the GPU-CPU and CPU-CPU bandwidth. For example, when GPU 0-3 send data to GPU 4-7 at the same time, the guaranteed bandwidth is at most 15.75 GB/sec (16 lanes of PCIe between the switch and the CPU) shared among them.

2.2 Data center

A data center hosts a cluster of computing machines and other components. It also provides software stacks to make the access to hardware more convenient to end users. Today, a large number of distributed computing jobs for machine learning are run in data centers. In this section, we give a brief overview of how hardware and software are structured in a typical data center. One may refer to [13] for more details.

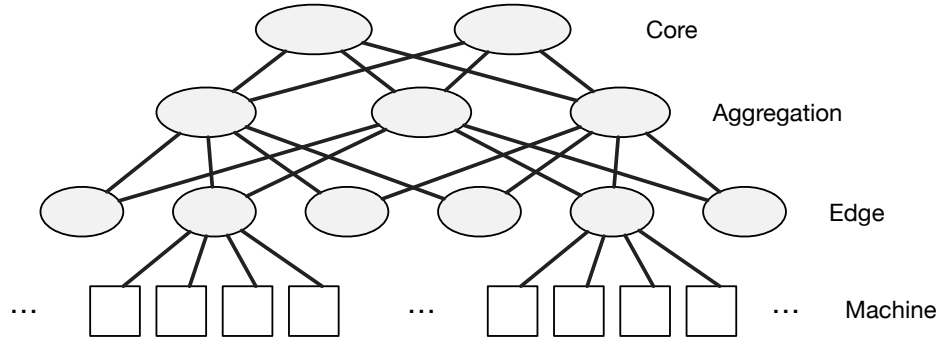


Figure 2.4: Multi-rooted tree topology for machine connections in a data center

In a data center, computing machines are connected in a network. A typical choice of the

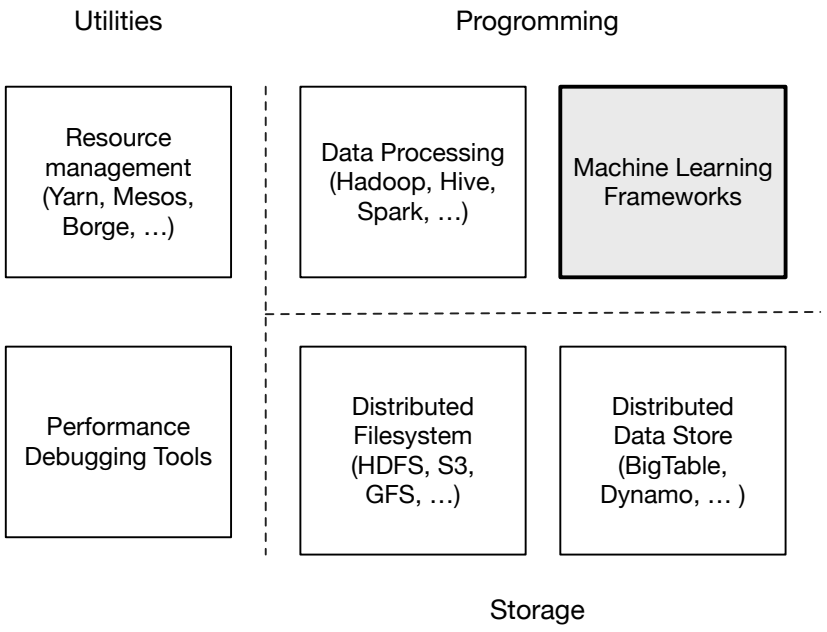


Figure 2.5: Cluster-level Infrastructure

network topology is the multi-rooted tree as illustrated in Figure 2.4. Machines are grouped into racks in a data center. Each rack contains tens of machines, which are connected via an edge switch. The edge switches are then connected by aggregation switches, which usually have a multi-layer structure. The last layer of aggregation switches are connected to the core data center switches.

Recall that in Figure 2.3 the communication bottleneck between two GPUs that belong to two different switches is the bandwidth of the single link between a switch and the CPU. Here, each edge switch has more than one up-links. Assume all links have the same bandwidth and an edge switch has four down-links to the four machines in a rack and two up-links to other switches, then when all four machines attempt to communicate to machines in other racks, each of them enjoys the bandwidth of half of an up-link. The ratio between the number of down-links and up-links is called the over-subscription ratio, which is 2:1 in this example, and is 4:1 in the example in Figure 2.3. An ideal over-subscription would be 1:1, also called full bisection bandwidth. Such ideal hardware configuration is very expensive.

There is an architecture for different software that are running in a data center. Usually the software can be classified into three layers [13]. The firmware, kernel, and operating systems that abstract the hardware of the computing machines run on the bottom level. The top level is the application level, where the data center runs the applications, which implement specific services such as hosting website, e-mail services, and various data processing jobs. Between the bottom level and top level there is the cluster level that manages the computing resources in the cluster and provides storage and compute services for the application level.

In Figure 2.5, we list several well-known infrastructures for the cluster level. They can be classified into three categories:

Resource managers abstract away the hardware, including CPU, GPU, memory and storage, from the physical machines to the application level. They allocate and manage resources for other software running in the cluster. Examples of resource managers include Yarn [52], Mesos [67], and Kubernetes [35].

The performance debugging tools are used to identify the performance bottleneck, which could be caused by inefficient usage of CPU, memory, disk, network.

Storage software provide storage services for shared data.

A distributed file system is a file system that provides an interface for machines to mount, list, read and write data. It can be mounted by multiple machines, and it can span over multiple machines for larger capacity. It often duplicates data in the storage and use the redundancy to achieve higher throughput. Examples of distributed file systems include GFS [57] and HDFS [53],

A distributed data store serves to enforce different APIs for storing structured data. For example, Dynamo [43] stores data by key-value pairs, BigTable [29] uses a sparse multi-dimensional sorted mapping as the data format, and Cassandra [54] provides a database interface.

Programming software are platforms that facilitate the end users to develop distributed computing programs.

A data processing framework utilize multiple machines to process large scale data sets. Notable examples include MapReduce [41] and its follower Hadoop [53] and Spark [152]. In general, a MapReduce program is composed of a Map procedure for data filtering and sorting, and a Reduce procedure for data summary operation. Another example of data processing frameworks is Flink [55], which supports streaming data flow.

A machine learning framework is specialized for different classes of machine learning algorithms, and it works closely with other related software. For example, a machine learning framework needs to communicate with the resource managers to submit computing jobs; it usually reads data directly from a distributed file system or a data store; and it may obtain the output from data processing frameworks.

Machine learning frameworks are the focus of the first part of this thesis. We propose two machine learning frameworks called Parameter Server (PS) (Chapter 3) and MXNet (Chapter 4).

Chapter 3

Parameter Server: Scaling Distributed Machine Learning

3.1 Introduction

Since its introduction, the Parameter Server (PS) framework [123] has proliferated in both academia and industry. In this chapter, we introduce our implementation of the third generation Parameter Server. The focus is on the system aspects of distributed inference, and our design decisions were guided by the workloads found in real systems.

3.1.1 Engineering Challenges

When solving distributed data analysis problems, the issue of reading and updating model parameters shared between different worker nodes is ubiquitous. The PS framework provides an efficient mechanism for aggregating and synchronizing statistics including model parameters between the worker nodes. In PS, each worker node only needs to maintain a small part of the model parameters which it typically operates on. Two key challenges arise in constructing a high performance PS system:

Communication In a conventional datastore, the parameters could be updated as key-value pairs. However, using this abstraction naively is inefficient: the values are typically small (floats or integers), thus the overhead of sending each update as a key-value pair can be very high.

Our insight to improve this situation comes from the observation that many learning algorithms represent parameters as structured mathematical objects, such as vectors, matrices, or tensors. At each logical time (or an iteration), typically only a small part of the object is updated. That is, worker nodes usually communicate just a *segment* of a vector, or a *row* of a matrix. This provides an opportunity to automatically batch process both the communication of updates and their processing on the PS, and allows the consistency tracking to be implemented efficiently.

Fault tolerance As argued before, for a distributed computing system, fault tolerance is a critical property especially at large scale. In particular, for efficient operation, upon failure of

	Shared Data	Consistency	Fault Tolerance
Graphlab [91]	graph	eventual	checkpoint
Petuum [40]	hash table	delay bound	none
REEF [32]	array	BSP	checkpoint
Naiad [102]	(key,value)	multiple	checkpoint
Mlbase [75]	table	BSP	RDD
Parameter Server	(sparse) vector/matrix	various	continuous

Table 3.1: Attributes of distributed data analysis systems.

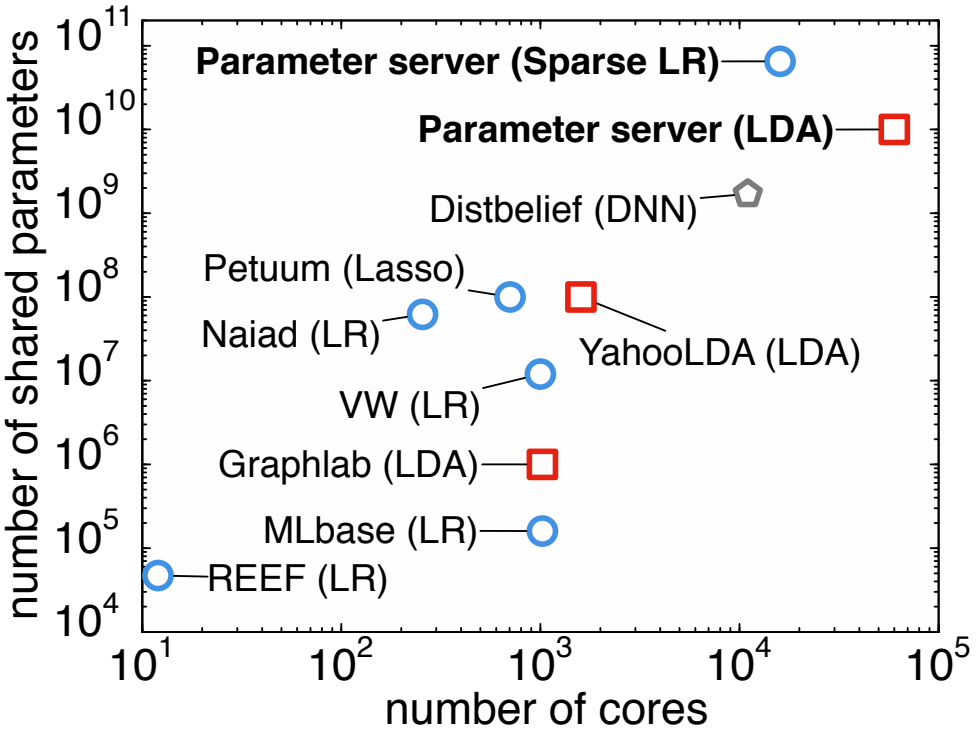


Figure 3.1: Largest machine learning experiments conducted using different computing systems. Problems: blue circles — sparse logistic regression; red squares — latent variable graphical models; grey pentagons — deep networks.

a single worker node it should not require a full restart of a long-running computation. To boost fault tolerance in PS, we implement live replication of parameters between servers to support hot failover, namely switching to a redundant node during running upon a node failure. Moreover, by treating machine removal or addition as failure or repair respectively, failover and self-repair in PS can in turn support dynamic scaling.

3.1.2 Our contribution

Our PS confers two main advantages to developers: first, by factoring out components of machine learning systems that are commonly required, it allows the application-specific codes to remain concise; second, as a shared platform targeting at system-level optimization problems, it provides a robust, versatile, and high-performance implementation, which is capable of handling a diverse array of algorithms ranging from sparse logistic regression to topic models and distributed sketching. In particular, our PS features the following five key properties:

Efficient communication We adopt the asynchronous communication model which does not block computation unless requested. Moreover, it is optimized for machine learning tasks to further reduce network communication overhead.

Flexible consistency models Our relaxed consistency further lowers synchronization cost and latency. Also, it offers the choice to balance the algorithmic convergence rate and system efficiency to the developers.

Elastic Scalability New computing machines and worker nodes can be added to the system without restarting the running framework.

Fault Tolerance and Durability Non-catastrophic machine failures can be repaired within 1s, without interrupting the computation process. Vector clocks ensure that the post-failure behaviors are well defined.

Ease to Use In order to facilitate the development of machine learning applications, the globally shared parameters are represented as potentially sparse vectors and matrices, which come with high-performance multi-threaded libraries.

Our PS is the first general purpose machine learning computing system that is capable of handling large scale problems at the industrial level. The novelty of the proposed system lies in the synergy of picking the right techniques of computing systems, adapting them to the machine learning algorithms, and modifying the machine learning algorithms to be more system-friendly.

Figure 3.1 provides an overview of the performance of the largest supervised and unsupervised machine learning experiments that are run on a number of well-known systems. When possible, we confirmed the scaling limits with the authors of each of these systems (data current as of 4/2014). As shown in the figure, we are able to handle orders of magnitude more data on orders of magnitude more processors than other published systems. Furthermore, Table 3.1 provides an overview of the main features of several distributed computing systems. Among them, our PS offers the greatest degree of flexibility in terms of consistency; it is the only system with continuous fault tolerance; and the type of its shared data makes it particularly user-friendly for data analysis applications.

3.1.3 Related Work

Related distributed computing systems have been implemented at large companies including Amazon, Baidu, Facebook, Google [42], Microsoft, and Yahoo [6], and there exist open source codes such as YahooLDA [6] and Petuum [68]. Graphlab [91] also supports parameter synchronization on a best effort model.

There were several major breakthroughs in the history of the development of Parameter Server. The first generation of PS, as introduced by [123] in 2010, lacked flexibility. In particular,

it repurposed `memcached` distributed (key,value) store as the synchronization mechanism. In 2012, YahooLDA improved this design by implementing a dedicated server with user-definable update primitives (set, get, update) and a more principled load distribution algorithm [6]. This second generation of *application specific* PS can also be found in Distbelief [42] and the synchronization mechanism of [82]. A first step towards a general platform was undertaken by Petuum [68] in 2013. This system improves YahooLDA with a bounded delay model while placing further constraints on the worker threading model. We will show that our third generation PS overcomes these limitations.

Finally, it is useful to compare PS to more general-purpose distributed computing systems for machine learning. Several of those systems can scale well to tens of worker nodes. However, since they mandate synchronous and iterative communication, at large scale, this synchronization significantly increases the chance of a worker node operating slowly. Mahout [12], based on Hadoop [53] and MLI [124], based on Spark [153], both adopt the iterative MapReduce [41] framework. A key insight of Spark and MLI is preserving state between iterations, which is indeed a core goal of PS.

We would also like to compare our PS with two other systems Distributed GraphLab [91] and Piccolo [111]. Distributed GraphLab [91] schedules its communication using a graph abstraction in an asynchronous way. However, GraphLab lacks the elastic scalability of the map/reduce-based frameworks, and it relies on coarse-grained snapshots for recovery, which also impedes scalability. Its applicability for certain algorithms is limited by its lack of global variable synchronization as an efficient first-class primitive. In a sense, a core goal of the PS framework is to capture the benefits of GraphLab's asynchrony and to go beyond its structural limitations. Piccolo [111] uses a strategy similar to PS to share and aggregate state between machines. In Piccolo, worker nodes pre-aggregate the state locally and transmit the updates to a server that keeps the aggregate state. It implements largely a subset of the functionality of our system, while lacking the optimization techniques specialized for machine learning, including message compression, replication, and variable consistency models expressed via dependency graphs.

3.2 Architecture

In this section, we discuss the architecture of our third generation PS.

As shown in Figure 3.2, the worker nodes are grouped into a server group and several worker groups. In the server group, one server node keeps track of the partition of the globally shared parameters; different server nodes communicate with each other to replicate and to migrate parameters for reliability and scaling; and a server manager node maintains a consistent view of the metadata of the server group, such as node liveness and the assignment of parameter partition.

Note that an instance of a PS can run more than one algorithm simultaneously. In particular, each worker group runs an application. There is a scheduler node in each worker group, which assigns tasks to the worker nodes in the group and monitors their progress. If workers are added or removed, the scheduler node is also in charge of rescheduling the unfinished tasks. A worker in a worker group typically only stores a portion of the training data locally to compute the local statistics such as the gradients. Workers do not communicate among themselves, and they communicate only with the server nodes to update and retrieve the shared parameters. Note that

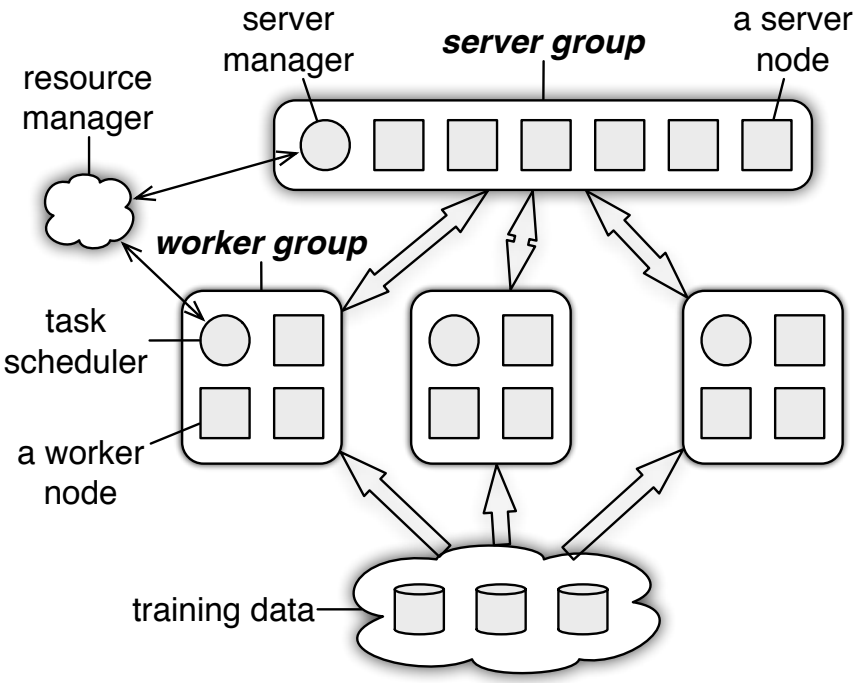


Figure 3.2: Communication between several groups of workers in Parameter Server.

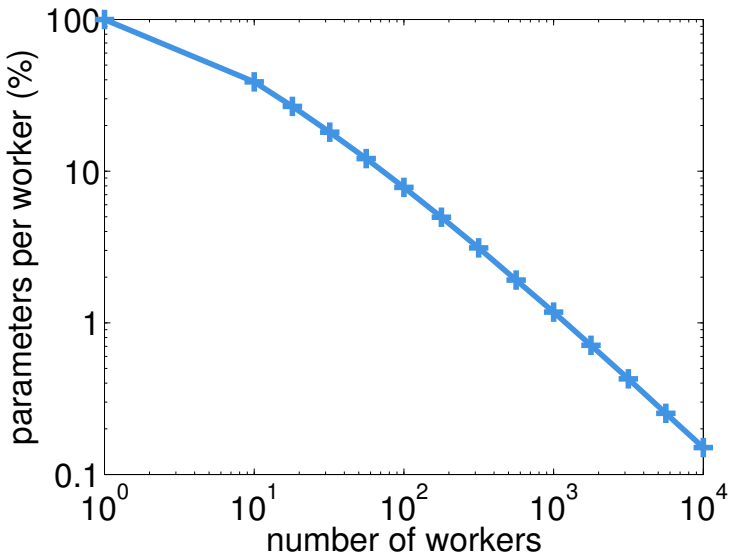


Figure 3.3: Parameters per worker node decreases with the number of workers.

each worker only needs the working set of the parameters for local computation, which is a small portion of all the parameters.

Figure 3.3 shows that with 100 workers, each worker only needs 7.8% of the parameters for the Ad click prediction application that will be discussed in detail in Section 6.3.1. With 10,000

Algorithm 2 Distributed Gradient-based Optimization with PS

Task Scheduler:

for iteration $t = 0, \ldots, T$ **do**
  choose $I_t \subseteq \{1, \ldots, n\}$
  partition $I_t = \bigcup_{k=1}^{m} I_{t_k}$
  issue WORKERITERATE(t) to all workers.
end for

Worker $k = 1, \ldots, m$:

function WORKERITERATE(t)
  pull $w_t^{(k)}$ from servers
  compute $g_t^{(k)} \leftarrow \sum_{i \in I_{t_k}} \partial f_i(w_t^{(k)})$
  push $g_t^{(k)}$ to servers
end function

Servers:

function SERVERITERATE(t)
  aggregate $g_t \leftarrow \sum_{k=1}^{m} g_t^{(k)}$
  use the gradient g_t to update w_{t+1}
end function

workers, this fraction further reduces to 0.15%, which leads to less memory required for each machine.

Our PS supports independent parameter namespaces, which allows a worker group to isolate its set of shared parameters from others. Several worker groups may also share the same namespace. One example of using this feature is that we can use more than one worker group to solve the same deep learning application [42] to achieve a higher level of parallelization. Another example is that a model can be actively queried by some worker nodes, in applications such as online services, while at the same time the model can be updated by a different group of worker nodes when new training data arrives.

Next, we discuss the key components of our parameter server in detail. The rest of this section unfolds as follows. The shared parameters are presented as (key,value) vectors to facilitate linear algebra operations (Section 3.2.1), and they are distributed across a group of server nodes (Section 3.3.3). Moreover, any node can both push out its local parameters and pull parameters from remote nodes (Section 3.2.2). By default, the tasks are executed by worker nodes, or they can also be assigned to server nodes via user defined functions (Section 3.2.3). The tasks are run in parallel and in an asynchronous way (Section 3.2.4). The PS provides the algorithm designer with the flexibility in choosing a consistency model via the task dependency graph (Section 3.2.5) and also in choosing the subset of parameters to communicate (Section 3.2.6).

As a toy example, in Figure 3.4, we visualize the PS implementation of Algorithm 2, the standard distributed gradient descent.

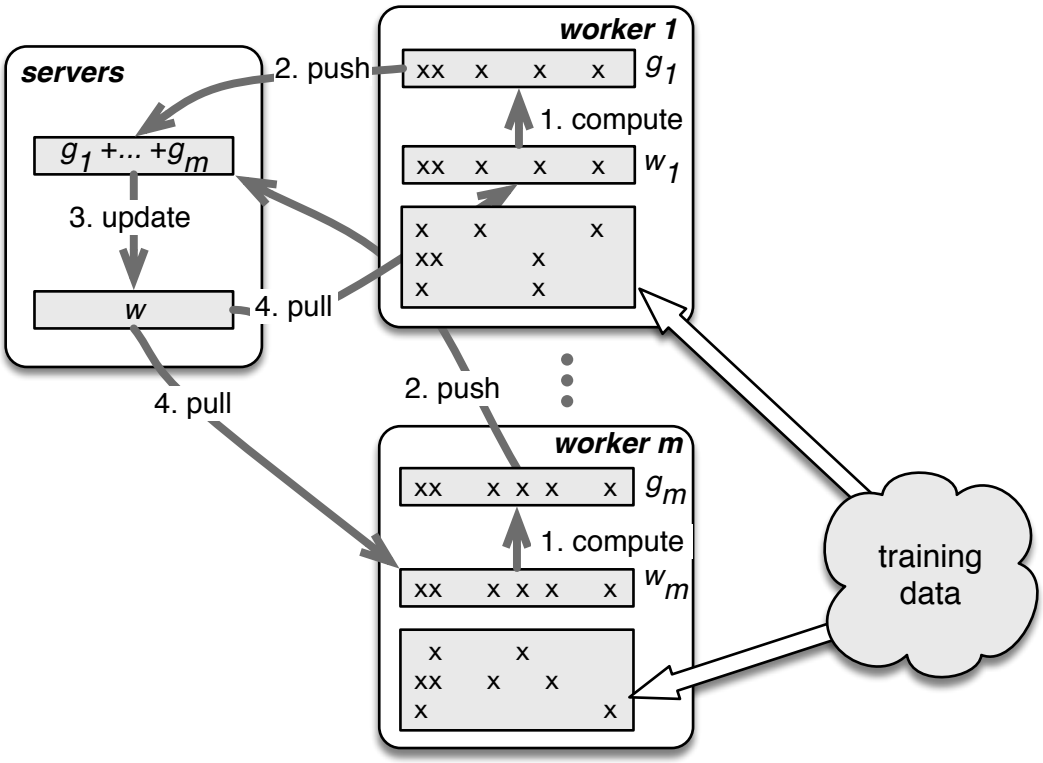


Figure 3.4: Steps of Algorithm 2. Note that each worker node only caches its working set of parameters w.

3.2.1 (Key,Value) Vectors

The model parameters shared among the worker nodes can be represented as a set of (key, value) pairs. For example, in a loss minimization problem, the pair is a feature ID and its weight. In LDA, the pair is a combination of the word ID and topic ID, and the frequency count. Each entry of the model parameters can be read and written locally or remotely accessed via its key. This (key,value) abstraction is widely adopted by existing approaches [40, 75, 102].

Our PS improves upon this basic approach by acknowledging the underlying meaning of these key-value items especially in machine learning algorithms, where the model parameters are typically in the form of a linear algebra object. For instance, for the risk minimization problem with the objective function in Algorithm 2, the model parameter w is a vector. Our PS can provide the same functionality as the (key,value) abstraction. Moreover, by treating these model parameters as linear algebra objects which are often sparse, it allows efficient implementation of important operations such as vector addition $w + u$, multiplication Xw, finding the 2-norm $\|w\|_2$, and other more sophisticated operations [48].

We assume that the keys are ordered, so that we can provide the vector and matrix semantics to the (key,value) pairs in the model parameters (non-existing keys are associated with zeros by default). This introduction of linear algebra in machine learning into the PS greatly reduces the effort of implementing the optimization algorithms. Moreover, this interface design also leads

to highly efficient codes by leveraging CPU-efficient multithreaded self-tuning linear algebra libraries such as BLAS [48], LAPACK [10], and ATLAS [146].

3.2.2 Range-based Push and Pull

In our PS, communication between nodes is through `push` and `pull` operations. In Algorithm 2 each worker node pushes the gradients it computes with local data to the server nodes, and then pulls back the updated weights. Our PS supports *range-based* push and pull, which bulk communication API greatly improves the computation and communication efficiency. For example, let $\mathcal{R}$ be the range of the keys, the command $\texttt{w.push}(\mathcal{R}, \texttt{dest})$ sends all existing entries in the model w whose keys fall in the range $\mathcal{R}$ to the destination `dest`, which can be either a particular node, or a node group. Similarly, the command $\texttt{w.pull}(\mathcal{R}, \texttt{dest})$ reads all existing entries of model w with keys in the range $\mathcal{R}$ from the destination `dest`. If we set $\mathcal{R}$ to be the whole key range, the whole model vector w is be communicated, and if we set $\mathcal{R}$ to be a single key, only an individual entry is chosen.

Note that this communication interface can be extended to communicate any local data structures that share the same keys as the model parameters. For example, in Algorithm 2, a worker pushes the gradients g it computes with local data to the PS for aggregation. Since the gradients g shares the keys of the model parameter set w at the worker node, the developer can simply use the command $\texttt{w.push}(\mathcal{R}, \texttt{g}, \texttt{dest})$ to push the local gradients.

3.2.3 User-Defined Functions on the Server

The nodes in the server group are not only in charge of aggregating data from worker nodes, but they can also execute user-defined functions. This is favorable as the server nodes often have more complete or up-to-date information about the shared model parameters. For example, in Algorithm 2, the server nodes use subgradients to update the model parameter w. In the context of sketching (see Section 3.4.3), almost all operations occur in the server nodes.

3.2.4 Asynchronous Tasks and Dependency

A task is issued by a remote procedure call, which can be a `push` or a `pull` that a worker sends to servers, or can also be a user-defined function sent by the scheduler to any node. Tasks may include arbitrary number of subtasks. For example, the task `WorkerIterate` in Algorithm 2 contains one `push` and one `pull`.

Tasks are executed asynchronously. The caller of the task can start other computation immediately after issuing a task without waiting for the reply from the callee. The caller marks a task as finished only when it receives the reply. A reply could be the function return of a user-defined function, or the (key,value) pairs requested by the `pull`, or an empty acknowledgement. On the other end, the callee marks a task it receives as finished only if the call of the task is returned and all the subtasks issued by this call are finished.

By default, callees execute tasks in parallel to achieve better efficiency. A caller that wishes to enforce serial task execution can place an dependency command *execute-after-finished* between different tasks. For example, figure 3.5 depicts three iterations of `WorkerIterate`. Iterations

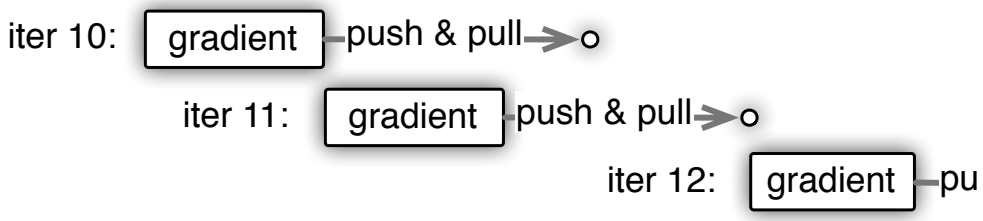


Figure 3.5: Example of asynchronous processing of different tasks by the same node. Here iteration 12 depends on 11, and iteration 10 and 11 are independent.

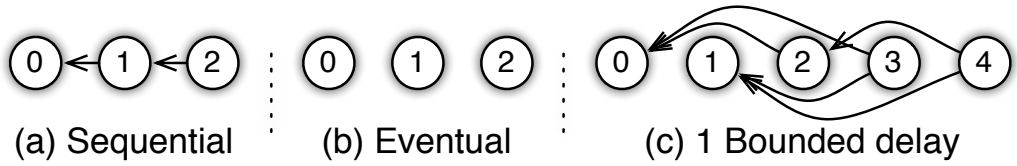


Figure 3.6: Directed acyclic graphs for different consistency models. The size of the DAG increases with the delay.

10 and 11 are independent, but 12 depends on 11. The callee therefore begins iteration 11 immediately after the local gradients are computed in iteration 10. Iteration 12, however, is postponed until the `pull` of 11 finishes.

Such task dependency facilitates implementing algorithm logic. For example, in Algorithm 2, the aggregation logic in `ServerIterate` updates the weight w only after when the gradients from all the workers are aggregated. This can be implemented by having the updating task depending on the push tasks of all workers. Another important use of task dependency is to support the flexible consistency models introduced next.

3.2.5 Flexible Consistency

It largely improve the efficiency of the system if independent tasks can be run simultaneously. This can be achieved by parallelizing the use of CPU, disk and network bandwidth. However, this may also lead to data inconsistency between nodes. For example, consider the diagram in Figure 3.5 for Algorithm 2, the worker starts iteration 11 before the updated model is pulled back, thus the outdated local model is used in iteration 11 and the same gradients are obtained as in iteration 10. This inconsistency can potentially slow down the convergence of the algorithm. Nevertheless, some algorithms are less sensitive to this type of inconsistency, and starting iteration 11 without waiting for 10 may only cause a small part of the model to be inconsistent.

The best trade-off between system efficiency and algorithm convergence usually depends on various factors, including the algorithm's sensitivity to data inconsistency, feature correlation of the training data, and the capacity difference between hardware components. Instead of forcing the user to adopt one fixed task dependency pattern that may be ill-suited to a specific problem, our PS gives the algorithm designers the flexibility in defining their own consistency models.

Figure 3.6 shows the directed acyclic graphs for three different models of task dependencies. Next, we discuss how each of these models can be implemented by our PS.

Sequential In sequential consistency, all tasks are executed one by one. The next task only starts after the previous one has finished. This consistency is identical to the single-thread implementation, which is also named as Bulk Synchronous Processing.

Eventual Eventual consistency is the opposite of sequential consistency, where all tasks start simultaneously. In [123] a system with eventual consistency is discussed. However, this consistency is only useful if the algorithms are robust to the possible delays.

Bounded Delay In bounded delay consistency, a maximal delay time τ is set. A new task is blocked until all the previous tasks started τ time slots ago are finished. This model provides more flexible controls than the previous two. Moreover it has the previous two models as special cases: setting $\tau = 0$ gives the sequential consistency model, and setting $\tau = \infty$ gives the eventual consistency model.

Note that the dependency graphs can evolve over time. The scheduler may increase or decrease the maximal delay according to the runtime progress in order to balance system efficiency and the algorithm convergence.

3.2.6 User-defined Filters

Besides the scheduler-based task flow control, our PS also supports various user-defined filters to selectively synchronize individual (key,value) pairs, allowing fine-grained control of data consistency within a task. The insight is that the optimization algorithm usually possesses information about which parameters are most useful and need to be synchronized with high priority. One example is the *significantly modified* filter, which only pushes entries that have changed by more than a threshold since their last synchronization. Another example is *fixed point quantization*, which converts float numbers into lower bit integers.

3.3 Implementation

In this section, we discuss the key implementation details of our PS. Different from prior (key,value) systems, our PS is optimized for *range based communication* with compression for both range based vector clocks (Section 3.3.1) and data (Section 3.3.2). Section 3.3.3 shows how the server nodes use consistent hashing [132] to store the key-value pairs of the model parameters. Section 3.3.4 shows that entries are replicated using chain replication [142] to achieve fault tolerance.

3.3.1 Vector Clock

In our PS, given the potentially complex task dependency graph and the need for fast recovery, each (key,value) pair is associated with a vector clock [43, 78], which records the time of each individual node on this (key,value) pair. Vector clocks have various benefits. For example, it is good for tracking aggregation status or rejecting doubly sent data. However, a naive implementation of the vector clock requires $O(nm)$ space to handle n nodes and m parameters.

Algorithm 3 Set vector clock to t for range $\mathcal{R}$ and node i

for $\mathcal{S} \in \{\mathcal{S}_i : \mathcal{S}_i \cap \mathcal{R} \neq \emptyset, i = 1, \ldots, n\}$ **do**
  if $\mathcal{S} \subseteq \mathcal{R}$ **then** $\mathrm{vc}_i(\mathcal{S}) \leftarrow t$ **else**
    $a \leftarrow \max(\mathcal{S}^b, \mathcal{R}^b)$ and $b \leftarrow \min(\mathcal{S}^e, \mathcal{R}^e)$
    split range $\mathcal{S}$ into $[\mathcal{S}^b, a), [a, b), [b, \mathcal{S}^e)$
    $\mathrm{vc}_i([a, b)) \leftarrow t$
  end if
end for

With thousands of nodes and billions of parameters, this is infeasible in terms of memory and bandwidth.

Fortunately, as a result of the range-based communication pattern of our PS, many parameters have the same time-stamp. In particular, if a node pushes the parameters within a range, then the time-stamps of those parameters are likely to be the same, and thus they can be *compressed* into a single range based vector clock. More specifically, let $\mathrm{vc}_i(k)$ denote the time-stamp of key k for node i. Given a key range $\mathcal{R}$, setting the ranged vector clock $\mathrm{vc}_i(\mathcal{R}) = t$ means that for any key k in the range $\mathcal{R}$ we set $\mathrm{vc}_i(k) = t$.

Initially, there is only one vector clock for each node i, which covers the entire key space and 0 is set to be its initial time-stamp. Each operation of range setting splits the range and may create at most 3 new vector clocks (see Algorithm 3). Let k be the total number of unique ranges defined by the algorithm, and let m be the number of nodes, then there are at most $\mathcal{O}(nk)$ vector clocks. Since k is typically much smaller than the total number of parameters, this range based vector clock significantly reduces the memory and bandwidth requirement[1].

3.3.2 Messages

When a node sends a message to an individual node or a node group, a message consists of a list of (key,value) pairs whose keys are in the range $\mathcal{R}$ and the associated range vector clock:

$$[\mathrm{vc}(\mathcal{R}), (k_1, v_1), \ldots, (k_p, v_p) \ : k_j \in \mathcal{R}] \,.$$

This is the basic communication format of our PS. It is for communicating both the shared parameters and the tasks. For the latter, a (key,value) pair usually assumes the form (task ID, arguments or return results).

Messages may only carry a subset of all the keys within range $\mathcal{R}$, and the missing keys are assigned the same time-stamp without changing their values. A message can be split by the key range. This happens when a worker sends a message to the entire server group, or when the key assignment of the receiver node has changed. To achieve this, we just need to partition the (key,value) lists and split the range vector clock similar to that in Algorithm 3.

Since machine learning problems typically require high bandwidth for communication between the nodes, it is desirable to compress these messages. One observation is that the training

[1]Ranges can also be merged to reduce the number of fragments. However, in practice both m and k are small enough to be easily handled. We leave range merging for future work.

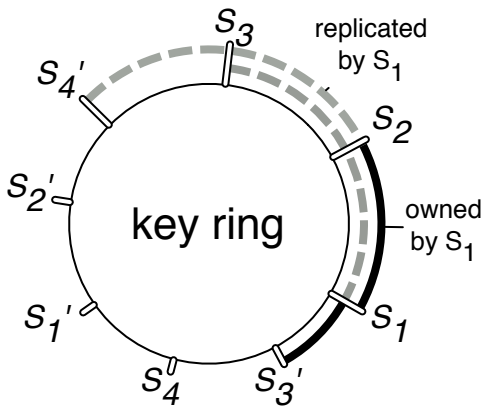


Figure 3.7: Server node layout.

data often remains unchanged between iterations of the algorithm, resulting that a worker sends the same key lists multiple times. Therefore, we allow the receiving node to cache the key lists, so that when such duplication happens later, the sender only needs to send a hash of the list. Another observation is that the model parameter may contain many zero entries. For example, a large portion of parameters remain zero in sparse logistic regression, as evaluated in Section 3.4.1. Likewise, a user-defined filter may also zero out a large fraction of the values. Hence we only need to send the (key,value) pairs with nonzero values. To achieve this, we use the Snappy compression library [63] to compress messages by effectively removing the zeros. Note that key-caching and value-compression can be used jointly in our PS.

3.3.3 Consistent Hashing

Similar to a conventional distributed hash table [24, 118], our PS partitions keys by inserting both keys and server node IDs into the hash ring (Figure 3.7). Each server node manages a key range, which starts with this server's insertion point and ends with the next insertion point of another node in the counter-clockwise direction. This node is called the master of this key range. A physical server is often represented in the ring via multiple "virtual" servers to improve load balancing and recovery.

We simplify the key range management by using a directly mapped DHT design. The server manager handles the ring management, and all other nodes locally cache the key partition, so that they can determine which server is responsible for a key range, and are notified of any change.

3.3.4 Replication and Consistency

Each server node stores a replica of the key ranges for the k counterclockwise neighbors on the ring relative to the one it owns. For example, in Figure 3.7, we have $k = 2$ and server 1 replicates the key ranges owned by server 2 and server 3. We refer to nodes holding copies of a key range as slaves of that key range, and the owner of the key range as the master of it.

Worker nodes communicate with the master of a key range for both `push` and `pull`. Any modification of a key range by the master is copied with the time-stamp to its slaves syn-

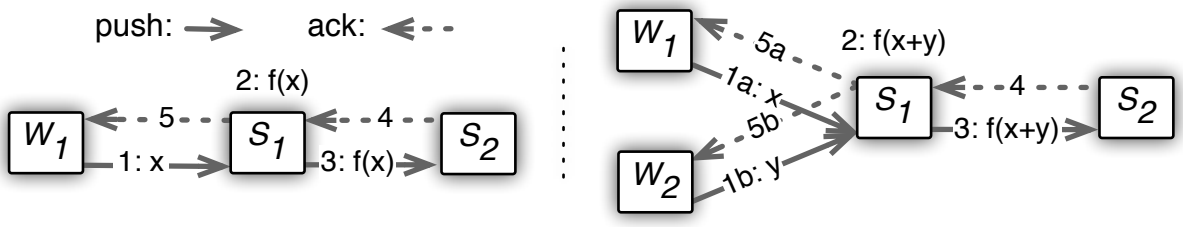


Figure 3.8: Servers generate replicas of key ranges. Left: a single worker. Right: multiple workers updating values simultaneously.

chronously. For example, on the left of Figure 3.8 it shows that when worker 1 pushes x into server 1, a user defined function $f(x)$ to modify the shared data is invoked. The push task is completed when the data modification $f(x)$ is copied to every slave.

Naive replication potentially increases the network traffic by k times, which is undesirable for many machine learning applications where the communication traffic is already high. To solve this problem, our PS framework permits "replication after aggregation". The basic idea is that the server nodes can first aggregate data from the worker nodes, for instance summing local gradients, and postpone replication until such aggregation is completed. For example, on the right-hand side of Figure 3.8, two workers push x and y to the server respectively. The server first aggregates the push by computing $x + y$, then applies the modification $f(x + y)$, and performs the replication at last. With n workers, such replication uses only k/n times the bandwidth. Often k is a small constant, while n can be hundreds or even thousands. Although one may argue that aggregation also increases the delay of the task reply, such latency can be reduced by our relaxed consistency conditions.

3.3.5 Server Management

Our PS supports dynamic addition and removal of nodes to achieve fault tolerance and dynamic scaling. In particular, the following steps happen when a server node joins:

1. The server manager assigns the new node a key range, which may come from a terminated node or from a key range split.
2. The new server node becomes the master of this key range, and it fetches data within this key range. The server node also gets the data corresponding to another k key ranges for which it serves as the slave.
   For the new server node, fetching the data in a key range $\mathcal{R}$ from some other node S proceeds similar to the Ouroboros protocol [110]. First, the node S pre-copies all (key,value) pairs in the range $\mathcal{R}$ together with the vector clocks. Note that this may cause a range vector clock to split similar to that in Algorithm 3. During the pre-copy stage, the node S updates with the new node all the changes that occurred in $\mathcal{R}$. Then, the node S stops accepting any message associated with the key range $\mathcal{R}$, whilex the new node takes over.
3. The server manager broadcasts the system change. Based on the key range changes, other

nodes shrink their data and resubmit the unfinished tasks to the new server node if needed. Upon receiving the broadcast message about the new server node and the key range change, a node first checks if it also maintains the key range. If so, and if this key range should no longer be maintained by it, this node deletes all the (key,value) pairs within this key range together with the vector clocks. Next, the node scans all the outgoing messages for which no reply has been received yet. If the key range of such a message overlaps with the new key range, the message is split according to the key range change and forwarded to the new server node. Such messages may be sent twice due to delays, failures, and lost Acks. Both the original recipient and the new server node are able to correct it by checking the vector clocks.

The server manager detects node failure by a heartbeat signal. The departure of a server node (voluntary or due to failure) is treated similarly to that when a new server joins. The server manager simply assigns the key range of the leaving node to a different server node. Integration with a cluster resource manager such as Yarn [52] or Mesos [67] is left for future work.

3.3.6 Worker Management

The steps of adding a new worker node is similar to but much simpler than adding a new server node.

1. The task scheduler assigns the new worker node the data within a key range.
2. The new node loads the corresponding training data either from a network file system or from some existing workers. Note that since the training data is often read-only, there is no two-phase fetch, which is different from Step 2 when new server node joins the system. The new node also pulls the shared parameters from the server nodes.
3. The task scheduler broadcasts the system change. Other worker nodes free the corresponding training data if needed.

The task scheduler may start a replacement when a worker leaves the system. We give the algorithm designer the option to continue without replacing a failed worker, for two reasons. First, if the training data is huge, compared to recovering a server node, recovering a worker node is usually much more expensive. Second, for large scale problems, losing a small amount of training data only has very limited effect to the overall optimization problem. Sometimes, it may even be desirable to actively terminate the workers that are too slow.

3.4 Evaluation

In order to evaluate the performance of our PS, we implement two representative algorithms: Sparse Logistic Regression for risk minimization, and Latent Dirichlet Allocation for generative models. To demonstrate the versatility of our approach, these experiments are run on clusters in two large Internet companies and a university research cluster.

	Method	Consistency	LOC
System A	L-BFGS	Sequential	10,000
System B	DBPG	Sequential	30,000
Parameter Server	DBPG	Bounded Delay KKT Filter	300

Table 3.2: Comparison of performance between Parameter Server and other systems.

3.4.1 Sparse Logistic Regression

Setting Sparse logistic regression is one of the most popular algorithms for large scale risk minimization [27]. It combines the logistic loss, i.e. $\ell(x_i, y_i, w) = \log(1 + \exp(-y_i\langle x_i, w\rangle))$, with the ℓ_1 regularizer, i.e. $\Omega(w) = \sum_{i=1}^{n} |w_i|$. The ℓ_1 regularizer encourages a compact solution of sparse vector, yet its non-smoothness makes the optimization problem difficult.

We use ad click prediction dataset CTRb with 170 billion examples and 65 billion unique features, which has a sheer volume of 636 TB uncompressed (141 TB compressed). We deploy our PS on a cluster of 1000 machines in CompanyA. Each machine is equipped with 16 physical cores, 192GB DRAM, and connected by 10 Gb Ethernet. Among the 1000 machines, 800 machines are worker nodes, and 200 are server nodes. During our experiment, this cluster is in concurrent use by other tasks.

Algorithm We propose and implement the DBPG algorithm in Chapter 6 for sparse logistic regression. This distributed algorithm differs from the standard version in the following ways. In each iteration, only a subset of parameters is updated. The worker nodes compute both the first order gradients and the diagonal part of the second order derivatives for this subset. Based on these aggregated local derivatives, in the parameter server, the server nodes update the model by solving a *proximal operator*. Moreover, to reduce communication overhead, we use a bounded-delay model over iterations, and we use a "KKT" filter to suppress transmission of parts of the generated gradient update with negligible magnitude.

Results We compare the performance of our PS with two special-purpose systems, named System A and B, both of which are developed by a large Internet company. The results are shown in Table 3.2. Notably, to perform the same computational task, our PS only requires 300 lines of code, while both Systems A and B consist of more than 10K lines of code. Our PS successfully reduces most complicated details of the distributed implementation of the algorithm into a generalized component that is reusable for many other algorithms.

We run sparse logistic regression in these three systems and compare the time each system takes to get the objective function to the same value.

Figure 3.9 shows that the relaxed consistency model substantially increases worker node utilization. As can be seen from the figure, workers in System A are idle for 32% of the time, and in system B, they are idle for 53% of the time, waiting for the barrier in each block. Our PS reduces this cost to under 2%. In PS, workers can start processing the next block without waiting for the previous blocks to be completed, and in this way we can alleviate the delay otherwise imposed by barrier synchronization. This gain does not come entirely free: we can observe

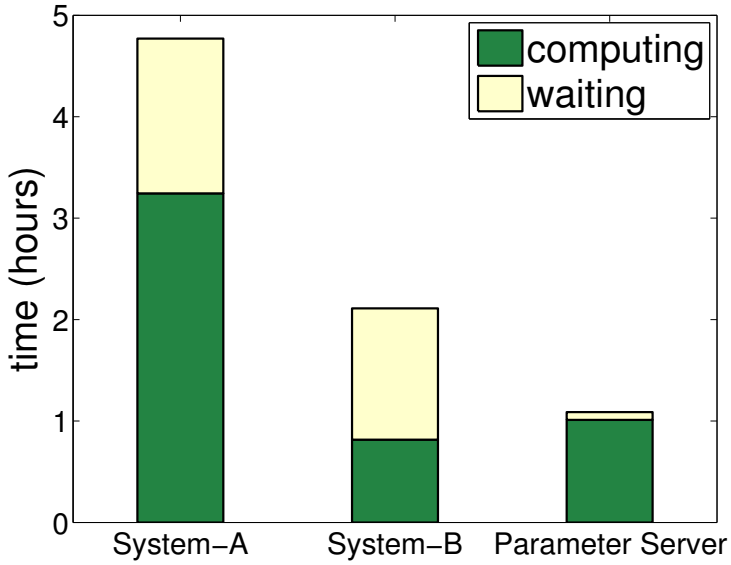


Figure 3.9: Time spent on computation and waiting (per worker) in sparse logistic regression.

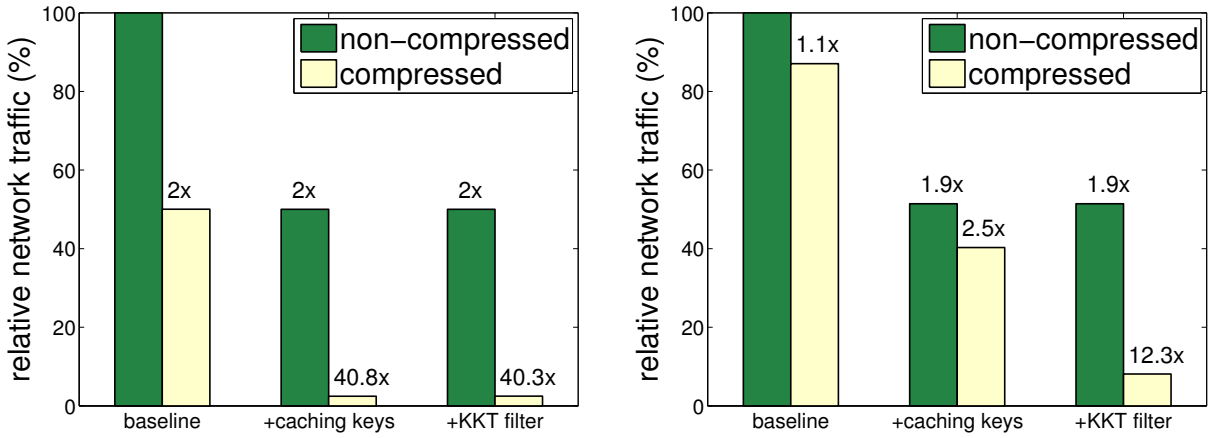


Figure 3.10: Savings of outgoing network traffic. Left: per server. Right: per worker.

that our PS uses slightly more CPU than System B. There are two reasons for this: first, and less fundamentally, System B optimizes its gradient calculations by careful data preprocessing; second, asynchronous updates with the PS require more iterations to achieve the same objective value. Nevertheless, thanks to the significantly reduced communication cost, our PS still halves the total runtime.

Figure 3.10 shows the reduction in network traffic. We can observe that allowing the senders and receivers to cache the keys helps to reduce nearly 50% of the traffic. This is because both key (`int64`) and value (`double`) are of the same size, and the key set is not changed during optimization. In addition, when applying the KKT filter, data compression is effective for compressing the values for both servers (>20x) and workers (>6x). The reasons for its effectiveness are twofold: first, the ℓ_1 regularizer encourages a sparse model, and thus most of values pulled from the servers are simply 0; second, the KKT filter also forces a large portion of the gradients

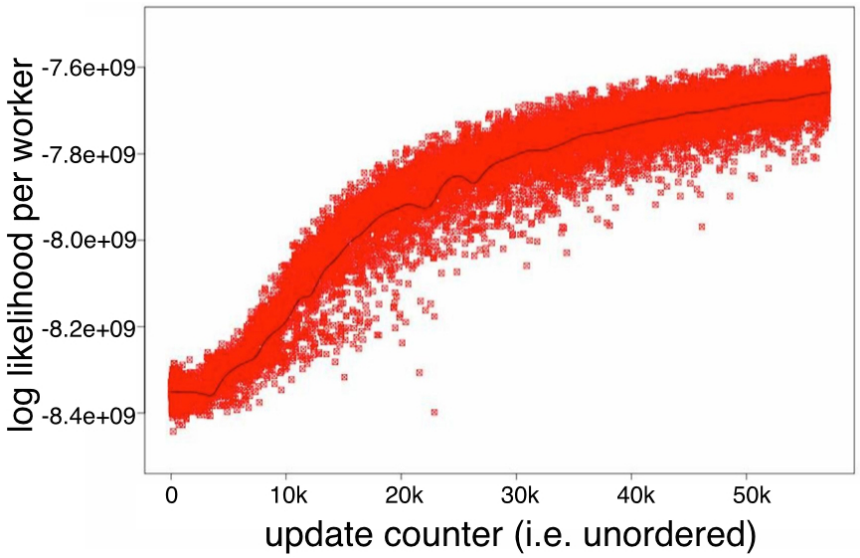


Figure 3.11: Distribution of log-likelihoods per worker as a function of time, in the setting of 1000 machines and 5 billion users.

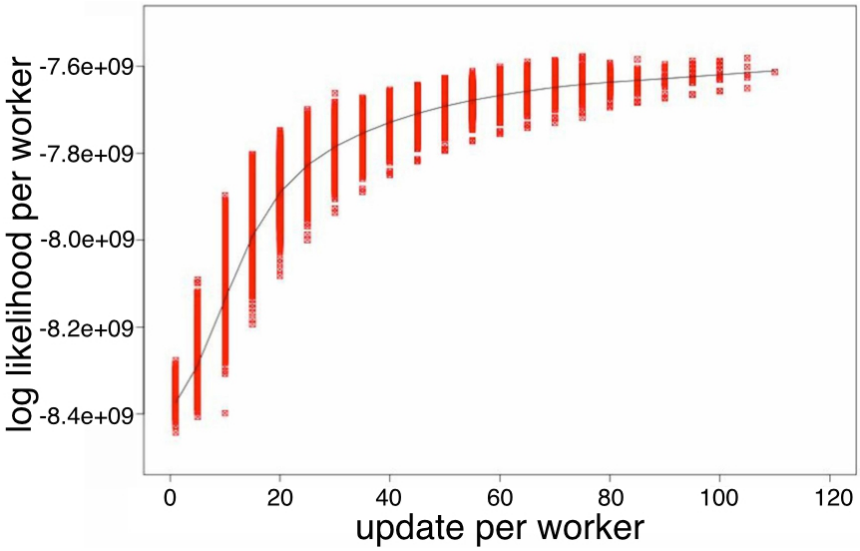


Figure 3.12: Distribution of log-likelihoods per worker as a function of time, stratified by the number of iterations.

sent to servers to be 0.

3.4.2 Latent Dirichlet Allocation

Setting We apply our PS architecture to solve the problem of modeling user interests based on the domain of the URLs which they click when presented with the search results. We collect search log data (ClickProfile in Table 1.4) with over 5 billion unique user identifiers and evaluate

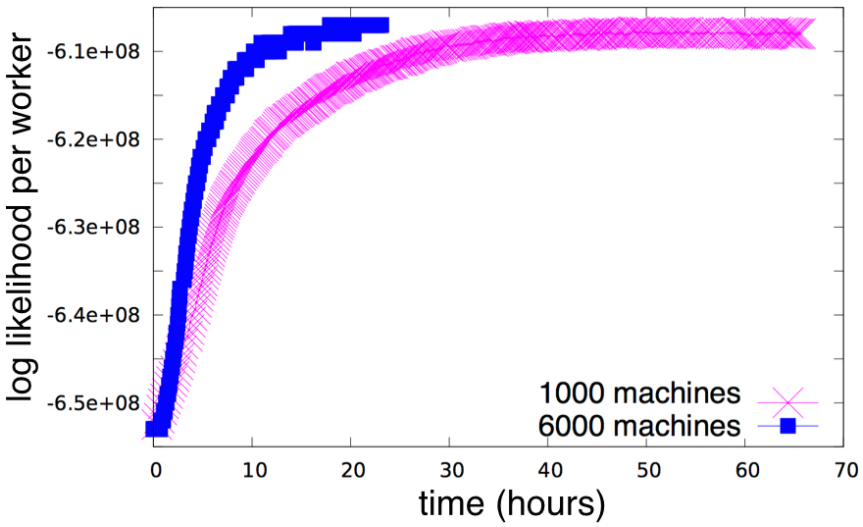


Figure 3.13: Convergence of log-likelihoods per worker, in the setting of 1000 and 6000 machines, 500 million users.

user interest model for the 5 million most frequently clicked domains. We run and compare the performance of the same algorithm with the setting of 800 workers and 200 servers, and with the setting of 5000 workers and 1000 servers respectively. The machines are from cluster CompanyB (Table 1.6). Each machine has 10 physical cores, 128GB DRAM, and at least 10 Gb/s of network connectivity. We again only have access the cluster with production jobs running concurrently.

Algorithm We performed LDA using a combination of Stochastic Variational Methods [69], Collapsed Gibbs sampling [62] and distributed gradient descent. Similar to the setting of [6], the gradients are aggregated asynchronously as they are sent from different worker nodes.

We divide the model parameters into local and global parameters. The local parameters (i.e. auxiliary metadata) are pertinent to a given node and they are streamed from the disk whenever we query a particular node. The global parameters are shared among the nodes and they are represented as (key,value) pairs to be stored using the PS.

To our knowledge, no other system (e.g., YahooLDA, Graphlab or Petuum) can handle this amount of data and model complexity for LDA, with up to 10 billion (5 million tokens and 2000 topics) shared parameters. The largest experiments [5] that have been previously reported only had under 100 million users active at any time, less than 100,000 tokens and under 1000 topics (2% the data, 1% the parameters).

Results To evaluate the quality of the inference algorithm with different settings, we use the training log-likelihood as a measure of fitting and monitor how rapidly it converges.

In Figure 3.13, we observe an approximately 4X speedup of convergence speed when the number of machines increases from 1000 to 6000. The stragglers observed in Figure 3.11 and 3.12 also illustrate the importance of having an architecture that can cope with performance variation across different worker nodes.

Algorithm 4 CountMin Sketch

Init: $M[i,j] = 0$ **for** $i \in \{1, \dots n\}$ **and** $j \in \{1, \dots k\}$. **Insert**(x)

1: **for** $i = 1$ **to** k **do**
2: $M[i, \text{hash}(i,x)] \leftarrow M[i, \text{hash}(i,x)] + 1$

Query(x)

1: **return** $\min \{M[i, \text{hash}(i,x)] \text{ for } 1 \le i \le k\}$

Peak inserts per second	1.3 billion
hline Average inserts per second	1.1 billion
Peak net bandwidth per machine	4.37 GBit/s
Time to recover a failed node	0.8 second

Table 3.3: Insertion rates of distributed CountMin implemented with Parameter Server.

3.4.3 Sketches

Setting To demonstrate the generality of our system, we test our PS with sketches [16, 37], which computation operates in a way that is very different from other machine learning algorithms: for sketches, a large number of write events constantly come from a streaming data source.

The computational task is to insert a streaming log of pageviews into a structure that allows the user to efficiently track pageview counts for a large collection of web pages. Each entry in the streaming log is a pair of the key for a webpage and the corresponding number of requests served in a particular hour. We use the Wikipedia (and other Wiki projects) page view statistics as a benchmark. For the time period between 12/2007 and 1/2014, there are 300 billion entries associated with more than 100 million unique keys. We run our PS with 90 virtual server nodes on 15 machines of a research cluster CampusB. Each machine has 64 cores and is connected by the Ethernet with 40Gb bandwidth.

Algorithm Sketching algorithms efficiently summarize and store the huge volumes of data so that the queries can be quickly answered. These algorithms are particularly important in streaming applications where a large volume of data and queries arrive in real-time. For example, Cloudflare's DDoS-prevention service analyzes the page requests across its entire content delivery service architecture to identify likely DDoS targets and attackers. The volume of data logged in such applications is far beyond the capacity of a single machine. While a conventional approach might be to shard a workload across a key-value cluster such as Redis, those solutions typically do not allow the user-defined aggregation semantics which are needed to implement *approximate* aggregation.

Algorithm 4 lists the key steps of the CountMin sketch algorithm [37]. In particular, a query returns an *upper* bound on the number of observed keys x. Splitting keys into ranges automatically allows us to parallelize the sketch algorithm. We thus implement a distributed CountMin with our PS.

Results Table 3.3 shows the insertion rates of distributed CountMin implemented with our PS. There are two key factors that contribute to the good performance. First, bulk communication featured in our PS significantly reduces the communication cost. Second, message compression further reduces the average (key,value) size to around 50 bits. Moreover, when a server node is terminated during the insertion operation, our PS can recover the failed node within 1 second. This fault tolerance makes our system well equipped for realtime applications.

Chapter 4

MXNet: a Flexible and Efficient Deep Learning Library

4.1 Introduction

4.1.1 Background

The term "deep learning" has come to represent a class of algorithms that can absorb huge volumes of data and learn elegant and useful patterns with that data. It prevails in various fields, including computer vision, natural language processing, and voice recognition. Thanks to the rapid growth of data volume and computing power, the developers are able to create complex neural network models, which can model the large scale data sets more accurately compared to conventional models. For example, almost all the recent ImageNet [119] challenge winners employ neural networks with hundreds of layers. These networks, however, require billions of floating-point operations to process each single sample, and there are usually millions to billions of samples in a typical large scale training data set.

The rise of modeling and computational complexity poses interesting challenges to the design of computing systems specialized for deep learning applications. We believe that an ideal system should meet the following criteria:

- **Scalability** Deep learning models can take days or weeks to train. Thus even modest improvements can make a huge difference in the speed at which new models are developed and evaluated. As one of the major themes of this thesis, distributed computing is a promising solution to speed up the training process by exploiting the computing power of different devices. The efficiency by which a deep learning framework scales out across multiple cores is one of its defining features.
- **Flexibility** Flexible and easy-to-use programming interface are also important. Deep neural network models are complicated and usually have a rich set of operators. Moreover, there are many ways to construct the objective function for the learning problems. The programming interface should allow users to specify the deep neural networks in a flexible way to accommodate all the variabilities. Besides, the developers also need to handle other common tasks such as data processing and visualization, which are supported by several

other specialized frameworks. In order to reduce the transferring cost, the programming interface of the deep learning system should either provide a similar interface to existing frameworks, or provide ways to interact with those frameworks.

- **Portability** Considering the trend of training and using deep learning applications in a wide range of situations — from laptops and server farms with high computing power, to mobiles and connected devices which often locate in remote locations with less reliable networking and considerably less computing power — portability to run on a broad range of devices and platforms is also required. However, efficient implementations of even the same algorithm on difference devices vary significantly. It is a great challenge to hide such implementation difficulties from the application developers.

A set of programming models have emerged to help developers define and train deep neural networks; along with open source frameworks that put deep learning in the hands of mere mortals. Well-known examples include Caffe [70], Torch7 [33], Theano [14], and TensorFlow [95].

All these systems embed a domain-specific language (DSL) into a host language (e.g. Python, Lua, C++). These programming paradigms can be classified into two categories: *imperative*, where the user specifies exactly "how" computation needs to be performed, and *declarative*, where the user specification focuses on "what" to do. In Table 4.1 we summarize the properties of the two paradigms. Examples of imperative programming include NumPy and Torch, while examples of declarative programming packages include Caffe and CNTK, which abstract away the inner-working of actual implementation and allow users to program over layer definition. The dividing line between the two categories can be muddy at times. Frameworks such as Theano and the more recent TensorFlow can be viewed as a mixture of both. These frameworks declare abstract computational graphs, while the computation within the graphs can be imperatively specified.

The learning bar of different deep learning systems is usually that programmers need to learn different host languages. For example, Caffe uses Protobuf as the host language while CNTK uses Brain Script. Despite the fact that the host language of many computing systems is Python nowadays, for most deep learning systems, programmers still need to learn a large set of new functions that are specific to systems.

4.1.2 Our contribution

To address the challenges in computing systems specialized for deep learning applications, we propose MXNet, a deep learning framework designed for both efficiency and flexibility. We highlight the key features of MXNet below:

Flexibility MXNet features a superset programming interface, which intends to blend the advantages of both imperative tensor computation and declarative symbolic expression. This allows the developers to enjoy the benefits of both approaches: declarative programming offers the clear boundary on the computation for discovering more optimization opportunities, whereas imperative programming has a much lower learning bar.

Well optimized back-end system . The user programs are transformed into an intermediate representation which is issued to the back-end system for execution. The back-end system uses technologies motivated from the compiler technology to optimize the memory

	Imperative Program	Declarative Program
Execute $a = b + 1$	Eagerly compute and store the results on a as the same type with b.	Return a computation graph; bind data to b and do the computation later.
Advantages	Conceptually straightforward, and often works seamless with the host language's build-in data structures, functions, debugger, and third-party libraries.	Obtain the whole computation graph before execution, beneficial for optimizing the performance and memory utilization. Also convenient to implement functions such as load, save, and visualization.

Table 4.1: Comparison between the imperative and declarative paradigm.

System	Core Lang	Binding Langs	Distributed	Imperative Program	Declarative Program
Caffe [70]	C++	Python/Matlab	×	×	√
Torch7 [33]	Lua	-	×	√	×
Theano [14]	Python	-	×	×	√
CNTK [149]	C++	Brain script/Python	√	×	√
TensorFlow [95]	C++	Python	√	×	√
MXNet	C++	Python/R/Scala/Julia	√	√	√

Table 4.2: Comparison between MXNet and other popular open-source ML libraries.

footprint and computation cost.

Efficiency MXNet has efficient implementation that achieve a high degree of scalability. We simplify the parameter server interface introduce in Chapter 3 for deep learning workloads. It explores the communication bandwidth hierarchy to best utilize the local communication in hetergeous computing to reduce the communication overherad.

In Table 4.2, we give a comparison between MXNet and other state-of-the-art computing systems.

The rest of this chapter unfolds as follows. We first introduce the front-end programming interface of MXNet in Section 4.2, and next in Section 4.3 discuss how the user programs are optimized and executed in the back-end systems. We explain how data are communicated between different devices in detail in Section 4.4. In Section 4.5 we show some encouraging experimental results to demonstrate its scalability. We conclude this chapter with some discussions of future work for MXNet.

4.2 Front-End Programming Interface

A deep neural network training program often consists of two parts: first defining the specifications of the neural network model; second specifying how the training algorithm interacts with the neural network, including how to load training data, how to update model parameters, and how to monitor the computation progress. The second part is common among machine learning

```
>>> import mxnet as mx
>>> a = mx.ndarray.zeros((2, 3), mx.gpu())
>>> a += 2
>>> b = mx.ndarray.dot(a, a.T)
>>> print(b.asnumpy())
[[ 12.  12.]
 [ 12.  12.]]
```

Figure 4.1: The NDArray interface in Python.

applications, while the first part is unique for deep learning applications. Given the complexity of deep neural network models, it is desirable that the programming interface has the flexibility to allow users to specify a complex model compactly and conveniently.

To address the need of flexibility, MXNet provides a multi-dimensional array computation interface to the users. Moreover, it allows symbolic expressions which make constructing neural networks more convenient. In the rest of this section, we explain these two interfaces. In the examples we use Python as the front-end language, but note that MXNet also supports other programming languages including R, Scala and Julia.

Multi-dimensional Array Computation `NDArray`, the library for multi-dimensional array computation in MXNet, is similar to that of Numpy, the most widely used scientific computing package in Python. NDArray provides multi-dimensional array manipulations and efficient linear algebra computations. Compared to Numpy, one major difference is that, beside CPU, NDArray also supports various computation devices such as GPU and FPGA. Figure 4.1 shows an example of doing matrix-constant multiplication on GPU using MXNet.

Symbolic Expressions A symbolic expression is an expression built from variables and operators. A symbolic variable is a free variable which can be assigned with value. A symbolic operator performs computations on one or multiple symbolic expressions and outputs new expressions. It can be a simple multi-dimensional array operation, such as element-wise addition, or a complex neural network component, such as the construction of a fully-connected layer. A symbolic expression only declares the computations, and all the free variables need to be bound with values when the expression is actually executed. This is a major difference compared to the multi-dimensional array interface, where all the variables are assigned with value upon initiation, with which the computation can be run step by step.

The neural network models can be constructed using symbolic expressions in MXNet. Figure 4.2 shows the construction of a multilayer perception by chaining the input data variable and several operators for layer construction. Note that even though we can write a deep learning program using only the multi-dimensional array interface, the availability of symbolic expressions still provides several advantages, especially for deep neural network construction. First, it is more convenient to save a symbolic expression in disks, which can be loaded using any front-end language supported by MXNet. Second, manipulating a symbolic expression incurs little overhead, which is useful when dealing with large scale deep neural networks. Third, all the computations

```
data = mx.symbol.Variable()
fc1  = mx.symbol.FullyConnected(data, num_hidden=64)
relu = mx.symbol.Activation(fc1, act_type='relu')
fc2  = mx.symbol.FullyConnected(relu, num_hidden=10)
mlp  = mx.symbol.SoftmaxOutput(fc2)
```

Figure 4.2: Example: define a multilayer perception using a symbol expression in MXNet.

```
mod = mx.mod.Module(mlp, mx.gpu())
mod.init_params(
    initializer=mx.init.Uniform(-1, +1))
mod.init_optimizer(
    optimizer='sgd',
    optimizer_params=(('learning_rate', 0.1),))
mod.forward(data=b)
mod.backward()
mod.update()
```

Figure 4.3: Example: create and run a module in MXNet.

are known before the program being executed, which makes it possible to optimize the training process before execution.

Computation Module Despite its efficiency and portability, declarative symbolic expressions are less straightforward for computation compared to imperative multi-dimensional arrays. MXNet provides `Module` to simplify the execution of symbolic expressions. A module works as a computation machine. It accepts symbolic expressions, with which variables it calls functions, such as forward to compute the output, backward to compute the gradients with respect to model parameters, and updating model parameters. Figure 4.3 shows an example of creating a module for the multi-layer perceptron defined in Figure 4.2 and running one iteration of stochastic gradient descent on GPU, using the multi-dimensional array defined in Figure 4.1 as the training data.

4.3 Back-End System

MXNet features a division between front-end programming interface and back-end system. In MXNet, the workloads created by front-end interface are pushed into backend system for optimization and execution. This division is motivated by how compilers works: a compiler transfers the source codes into an intermediate representation, which is then transferred and optimized to obtain the target program. The back-end system of MXNet is similar to a compiler. It first represent the the workloads, which are in the form of multi-dimensional array and symbolic expressions, as computation graphs, an intermediate representation of the system; then these

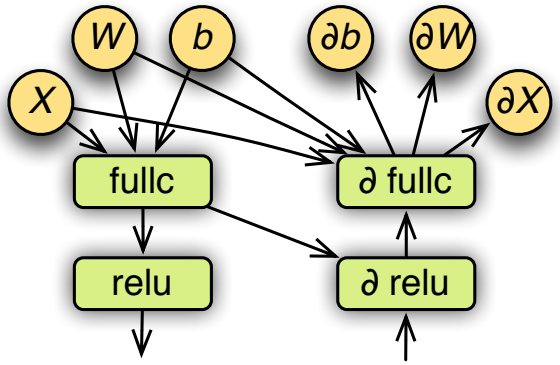


Figure 4.4: A partial computation graph for the forward and the backward of a fully connected neural network. Yellow circles and green rectangles represent data variables and operators, respectively. Arrows indicate data Dependencies between variables and operators.

computation graphs are transformed and optimized for more efficient execution on the target devices; finally, the back-end system schedules the execution of the computation graphs.

4.3.1 Computation Graph

Computation graph serves as the intermediate representation of workloads issued from the front-end to the back-end. A computation graph is usually defined as a directed bipartite graph, consisting of two sets of nodes which represent the variables and the operators. A variable is associated with a storage unit, and it can write the value to the storage unit and output the value it is assigned with. An operator defines the computation. It takes one or multiple variables as the inputs and generates one or multiple outputs. In particular, an input variable can be also used as the output variable. In a computation graph, an edge connecting node a to node b means that the variable a is the input or the output of operator b. For example, Figure 4.4 shows part of the computation graph of both the forward and the backward operation for the multilayer perceptron defined in Figure 4.2.

4.3.2 Graph Transformation and Execution

Before being executed, the computation graphs first go through some transformations, mainly for the following purposes. First, the graph received from the front-end may need additional information to be executed, such as allocating memory for variables and identifying which device to run an operator. Such necessary information can be added during graph transformation. Graph transformation also helps to reduce the memory footprint and facilitate graph optimization for more efficient execution. The back-end system of MXNet has a variety of techniques of graph transformation. Next, we discuss a few implementation details to illustrate the effectiveness of graph transformation.

Memory Optimization Naively allocating different memory units to different variables may result in a large memory working set, which is unfavorable for large scale computation. For a given computation graph, observe that the life time of each variable, namely the period between its initiation and the last time the variable is used, can be calculated before the execution. This suggests a more economical way of using the memory: variables that are non-overlapping in terms of their life time can share the same storage unit.

However, for a computation graph with n variables, to obtain the optimal memory allocation strategy has time complexity in the order of $O(n^2)$. Instead, we proposed two heuristics for finding non-overlapping variables in order to obtain a suboptimal memory allocation schemes. Both heuristics have time complexity linear in n. The first heuristic, called in-place, simulates the graph execution. This method keeps track of the lifetime of each variable, and the memory of a variable is recycled when the variable will not used by any operator later. The second heuristic, called co-share, finds the variables which can share the same memory space with the least conflicts. For more details of these heuristics, one can refer to our paper [31].

Fusion Operations Sometimes, the overhead of executing an operator, such as memory read / write and function call, may dominate the actual computation cost. In such situations, we combine several operators into a super operator to reduce the implementation overhead. For example, the normal computation graph for the function $(a \times b + c)$ involves two operators, namely $\times$ and $+$, and one temporary variable to store the intermediate result of $(a \times b)$. In the fused version, we can define a single operator which takes a, b and c as inputs and outputs the results directly. Besides eliminating the temporary variable and one function call, the fused operator can even use the fused multiplyadd instruction supported by the specific hardware to further decrease the number of instructions. Note that operator fusion usually consists of two steps: first, a subgraph is merged into a single fused node; then, the codes for the fused node are generated, which can be executed by runtime compilation later.

Lazy Evaluation After issuing the workloads in the form of computation graphs to the back-end system, the front-end thread returns immediately without waiting for the execution to be finished. We call this lazy evaluation. It allows the back-end system to batch process multiple graphs, which potentially improves the graph optimization and parallel execution. For example, to execute these two multi-dimensional arrays operations, $t = a \times b$ and $c = t + c$, with lazy evaluation we can merge these two into a single operator with fusion operation. Note that lazy evaluation does not affect the semantics of the programming interface. If the outputs of the operators are needed for other tasks such as copying to non-MXNet variables, the front-end would implicitly call the 'wait to read' function, which blocks the thread until the outputs are ready.

Parallel Execution The transformed and optimized computation graphs are then pushed into the back-end engine for execution. All the engine threads execute the operators in the graph in parallel as long as the graph dependencies are satisfied. Parallel execution is particularly important when there are multiple connected computation devices in a heterogeneous computing environment.

4.4 Data Communication

In a distributed computing setting, data are communicated between devices. How to reduce the communication overhead is a main issue addressed in this thesis. As deep learning applications usually involve large scale computation and are most suitable for distributed computing, in this section, we discuss the issue of data communication in MXNet for both the front-end interface and the back-end implementation.

The main feature of MXNet in terms of data communication is `KVStore`. This distributed key-value store module serves to reduce the data communication overhead. Compared to the low-level function `copyto` which is called to move data between devices explicitly, KVStore provides a high-level abstract interface for communication between devices.

4.4.1 Distributed Key-Value Store

KVStore maintains and updates a list of (key, value) pairs across multiple devices. The key can be an integer or a string associated with the value, which is often a multi-dimensional array. A device can either push a list of pairs into the key-value store or pull back the values corresponding a given list of keys. A customized update function can be registered into the store. The update function specifies how the pushed values are merged into the store. By default the "add to" updater is used, namely the new values are added in the store.

Figure 4.5 shows how to implement stochastic gradient descent (SGD) with `KVStore`. It differs from the implementation in Figure 4.3 in two ways. First, we set the SGD updater into the store instead of running the computation module. Second, we pull the recent parameters from the store before the forward pass, and then push the results, i.e. the gradients, into the store after the backward pass to update the model parameters.

When the codes in Figure 4.5 are executed using multiple computing devices located in different machines, KVStore sums over the gradients that are pushed by different devices before calling the update function. With this sequential data consistency model, any pull operation after the push operation is guaranteed to access the updated model parameters. Therefore, if we partition the data into m parts and assign them to the m devices, the results we obtain will be equivalent to that of running the same algorithm on a single device. This approach is often called data parallelism.

Besides the sequential data consistency model, KVStore also supports other relaxed data consistency model discusses in Chapter 3. For example, with the asynchronous consistency model, KVStore does not wait till all the gradients are received. It updates the model parameter immediately upon receiving the gradients from each device.

4.4.2 Implementation of KVStore

We implemented `KVStore` based on the parameter server architecture presented in Chapter 3. Here, we need two major modifications: first, since both push and pull are issued by the front-end just issues them as operation nodes to the back-end system, they can be further optimized for execution and do not need to be executed immediately. Second, since the device-to-device

```
kv = mx.kvstore.create()
def update(key, grad, weight):
    weight += 0.1 * grad
kv.set_updater(update)
for key, weight in enumerate(model.get_params()):
    kv.pull(key, weight)
mod.forward(data)
mod.backward()
for key, grad in enumerate(model.get_gradients()):
    kv.push(key, grad)
```

Figure 4.5: Run one SGD iteration with the key-value store.

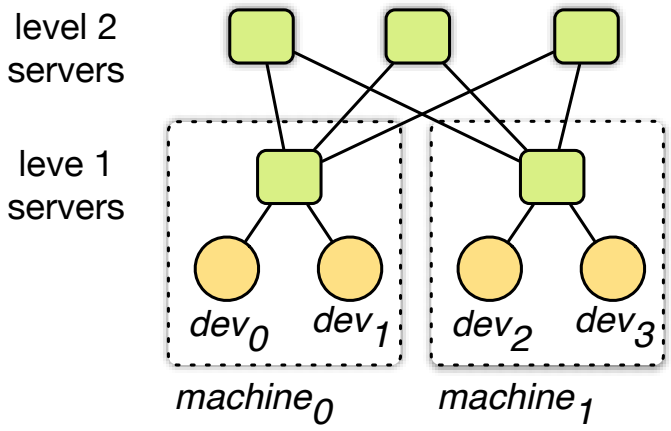


Figure 4.6: Two-level parameter server for KVStore. The level 1 server nodes aggregate data over devices on the same machine, and the level 2 server nodes communicate data between machines.

communication bandwidth within a machine can be 10 times higher than that of the machine-to-machine communication (see Section 2.1), we adopt a two-level server structure to better explore the communication hierarchy.

The two-level architecture is illustrated in Figure 4.6. Each machine may have multiple computing devices, and each device has one worker node. At level 1, each machine has one aggregation server node for communication between the devices on this machine. At level 2, there is another layer of server nodes that connect the aggregation server nodes from level 1. These server nodes are for communication between different machines. For example, when running SGD, the level 1 servers sum all data pushed by the devices on each machine before sending them to level 2 servers; similarly, level 1 servers pull back data and broadcast to the corresponding devices. In this way, the intra-machine traffic can be greatly reduced at the cost of latency caused by data aggregation within the machines.

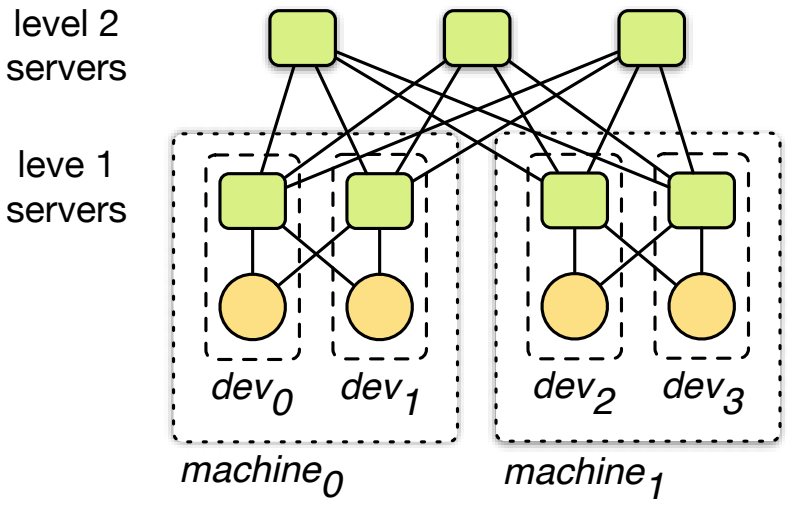


Figure 4.7: Two-level parameter server for KVStore, where each device has a level-1 server.

Recall that devices within a machine may have faster peer-to-peer communication bandwidth than device-to-CPU communication bandwidth. Figure 4.7 shows an architecture which leverages this property. Instead of associating a single server node with each machine, we run a server node in each device. In this setting, the communication between level 1 server nodes and worker nodes in the same machine is though peer-to-peer communication, which is much faster than device-to-CPU communication. Note that this is at the cost of more memory usage.

4.5 Evaluation

In this section, we provide experiment results to demonstrate the performance of MXNet. We shall focus on its computational efficiency in the distributed computing setting. Other experiments including the performance on a single device can be found in our paper [31].

For the training process, we run synchronized stochastic gradient descent (SGD) on the convolutional neural network model ResNet [65]. The ResNet we use has 152 convolutional layers. It has in total 157 data variables that need to be communicated between machines in distributed computing, and the total size of these variables is 240 MB.

All the experiments are run on the cloud service of AWS EC2 P2.18xlarge instances. Each instance is a virtual machine, and is equipped with 8 Nvidia Tesla K80 Accelerators. Each accelerator has two GPUs. We use the Deep Learning AMI version 1.4 from AWS marketplace, where CUDA 7.5 and cuDNN 5.1 are pre-installed. In terms of communication, there are two PCIe 3.0 16x switches, and each of the switches connects 4 K80 accelerators. The GPUs share the same PCIe switches to communicate with the CPU. All the machines are connected by the Ethernet with a bandwidth of 25 Gbit/sec.

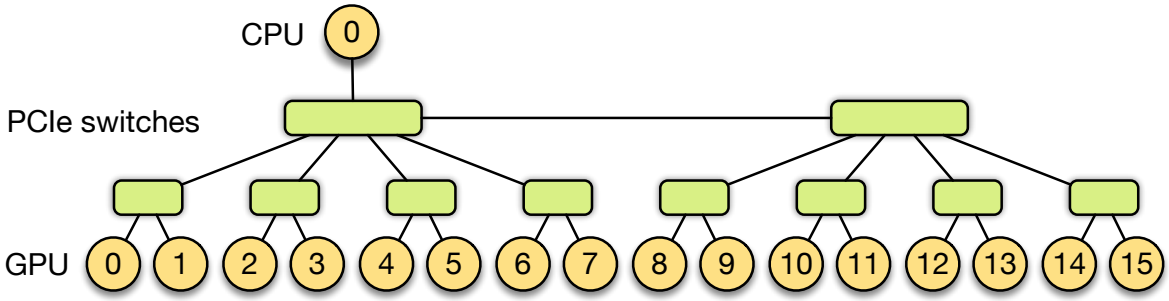


Figure 4.8: The topology of GPU connections for P2.16xlarge. Each line indicates a PCIe 16x connection.

4.5.1 Multiple GPUs on a Single Machine

We first show the results of implementing MXNet with multiple GPUs on a single machine (an AWS EC2 P2.18xlarge instance). We examine the communication cost, namely is the time spent on pushing and pulling synthetic data in each iteration, and the total cost, which is the total elapsed time during which the gradient computation, parameter updates, and data communication take place. For synchronized SGD, the batch size is the number of examples processed in each iteration. In our experiments, we fix the batch size of examples processed by every GPU, while increasing the number of GPUs from 1 to 16. Thus the total batch size of SGD increases with the number of GPUs.

Figure 4.9 shows the communication cost versus the total cost when running one iteration of SGD on ResNet-152.

The red line corresponds to the communication cost. Note that the communication cost scales almost linearly with the number of GPUs when the number of GPUs is in the range between 2 and 8. To run one SGD iteration with 8 GPUs, each worker node at a GPU sends 240MB of data variables, while the 8 server nodes send 240MB $\times$ 8 of data variables in total. In Figure 4.9, we observe a communication cost of 0.1s for 8 GPUs. This means that the uni-directional aggregate GPU bandwidth is at least 38.4 GB/sec for 8 GPUs, which achieves over 60% of the theoretical maximum uni-directional aggregated bandwidth 63GB/sec for the shared 4 PCI-e 16x connections. In Figure 4.9, there is a 4x increase of the communication cost when the number of GPUs increases from 8 to 16. The reason is twofold. First, each GPU can only communicate via peer-to-peer connection to at most 8 other GPUs. Assume that GPU 0 can communicate with GPU 1-8 via the PCIe switches directly. If GPU 0 needs to communicate with GPU 9-15, the communication has to be routed through the CPU, which incurs additional communication overhead. Second, even when peer-to-peer connection is available, for example when GPU 0 communicates with GPU 7, the single PCIe 16x connection between the two PCIe switches now becomes the bottleneck.

The other colored lines in Figure 4.9 correspond to the total cost for different batch size per GPU. For each line, since we fix the batch size per GPU, ideally the total cost of one SGD iteration stays the same as the number of GPUs increases. Namely the ideal scalability is linear

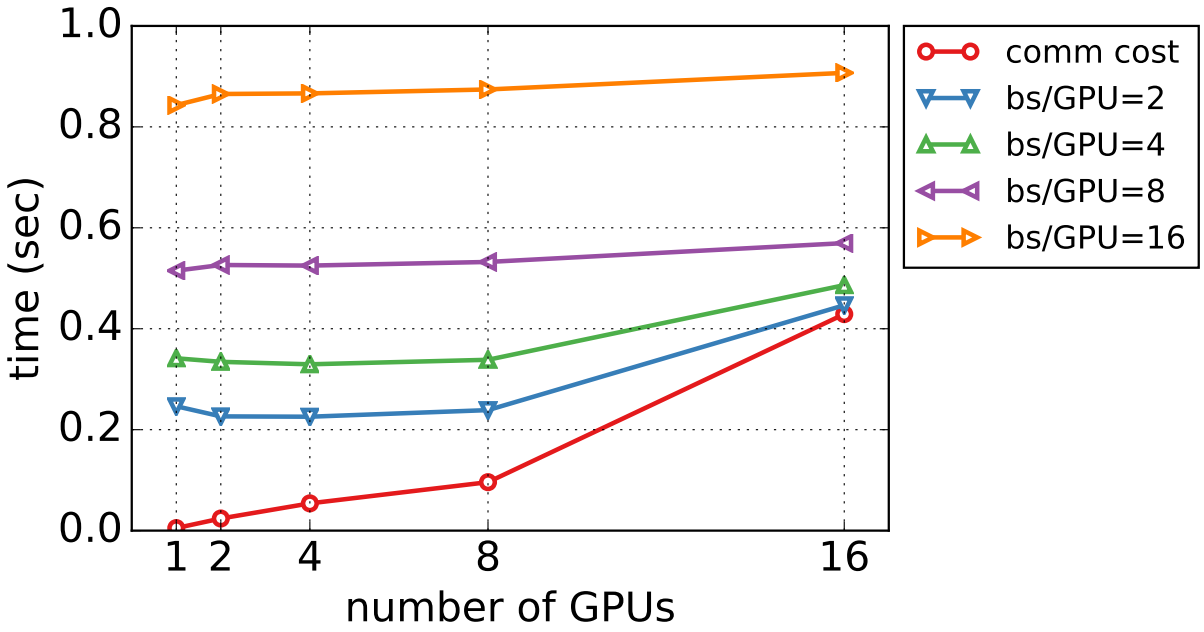


Figure 4.9: The communication cost and total cost of one SGD iteration on ResNet-152. Experiments are performed on a single machine, and Number of GPU = (1,2,4,8, 16).

in the number of GPUs. However, in practice the total cost may increase due to the increasing communication cost when the number of GPUs increases. In Figure 4.9, observe that when the number of GPUs increases from 1 to 8, all the lines stay nearly flat, namely we achieve a near-ideal scalability. This is due to the fact that in this range the communication cost is small and the total cost is dominated by the actual computational time. However, since the communication cost significantly increases when the number of GPUs is 16, we see that the total cost also slightly increases, especially for a small batch size per GPU.

4.5.2 Multiple GPUs on Multiple Machines

We also measure the communication cost and total cost of implementing MXNet on multiple machines. We plot the communication cost of one SGD iteration in Figure 4.10. In particular, for each colored line in the figure, we fix the number of GPUs used on each machine; and we plot the communication cost versus the number of total GPUs used when increasing the number of machines used as 2, 4, 8, 16.

The red line corresponds to the setting where we use one GPU on each machine. The communication cost increases linearly from 0.22s to 0.68s when the number of machines increases from 1 to 16. Compare the red line with the other lines for which we use 2, 4, 8 GPUs on each machine respectively. We observe that for a fixed number of machines used, the communication cost only increases by less than 0.1s when we increase the number of GPUs per machine. This is far less than the linear growth of each line for fixed number of GPUs per machine while the

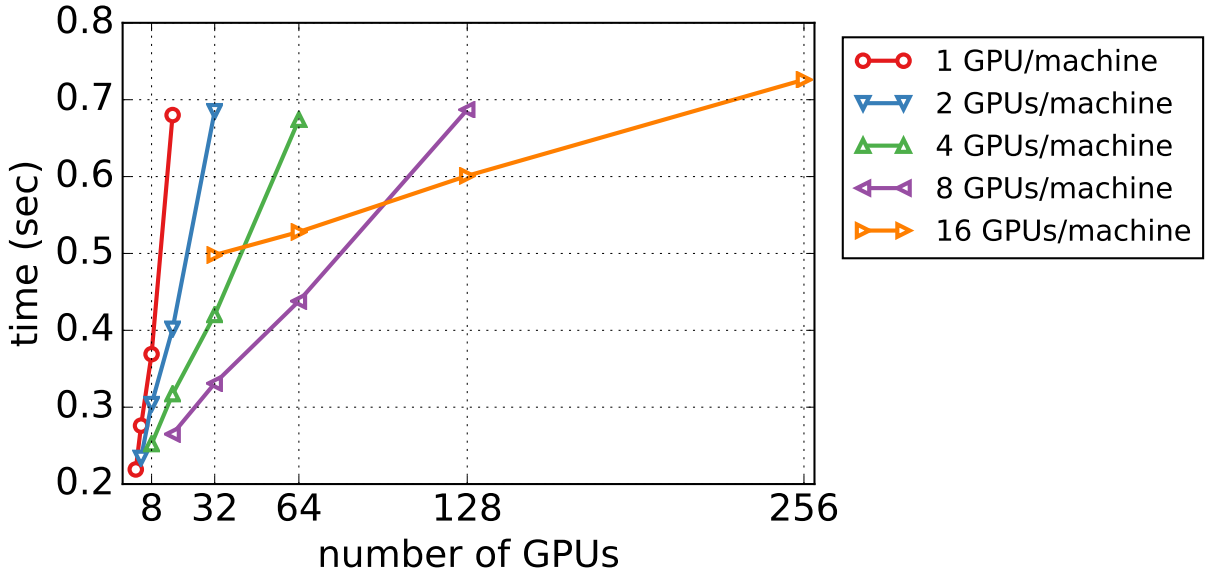


Figure 4.10: The communication cost of one SGD iteration for different number of machines (2,4,8,16) and different number of GPUs per machine (1,2,4,8,16).

number of machines increases as 2, 4, 8, 16. This comparison shows the efficiency of the 2 level parameter server architecture when there are more GPUs per machine.

In Figure 4.10, we also observe a large increase of communication cost when the number of GPUs is 16 (the orange line). This is similar to what we have observed in Figure 4.9. The lacking of full GPU peer-to-peer access requires a significant amount of shared CPU memory bandwidth, which slows down the data communication between machines.

In Figure 4.11, we plot the communication cost and total cost of one SGD iteration for increasing number of machines (1,2,4,8,16) with the number of GPUs per machine fixed to be 8. Different colored lines correspond to different batch size per GPU. Similar to our observation of Figure 4.9, for small batch sizes (bs/GPU=2,4,8), the total cost increases significantly with the number of GPUs, due to the fact that a large part of the total cost is the increasing communication cost. However, for a sufficiently large batch size (bs/GPU=16), the actual computation cost is much higher than the communication cost, and we observe near-linear scalability.

4.5.3 Convergence

In Section 4.5.1 and 4.5.2, we observed that a large batch size of SGD helps to amortize the communication cost and lead to a near-optimal scalability of the total cost. However in practice, a large batch usually slows down the algorithm convergence. In this section, we show some heuristics of tuning the SGD learning rate to improve the convergence speed when the batch size is large. A more detailed study can be found in Chapter 7.

In our experiments, we train ResNet-152 using the dataset Imagenet [45] As a baseline, we use 8 GPUs with batch size per GPU 32, namely an aggregate batch size 256 for the algorithm.

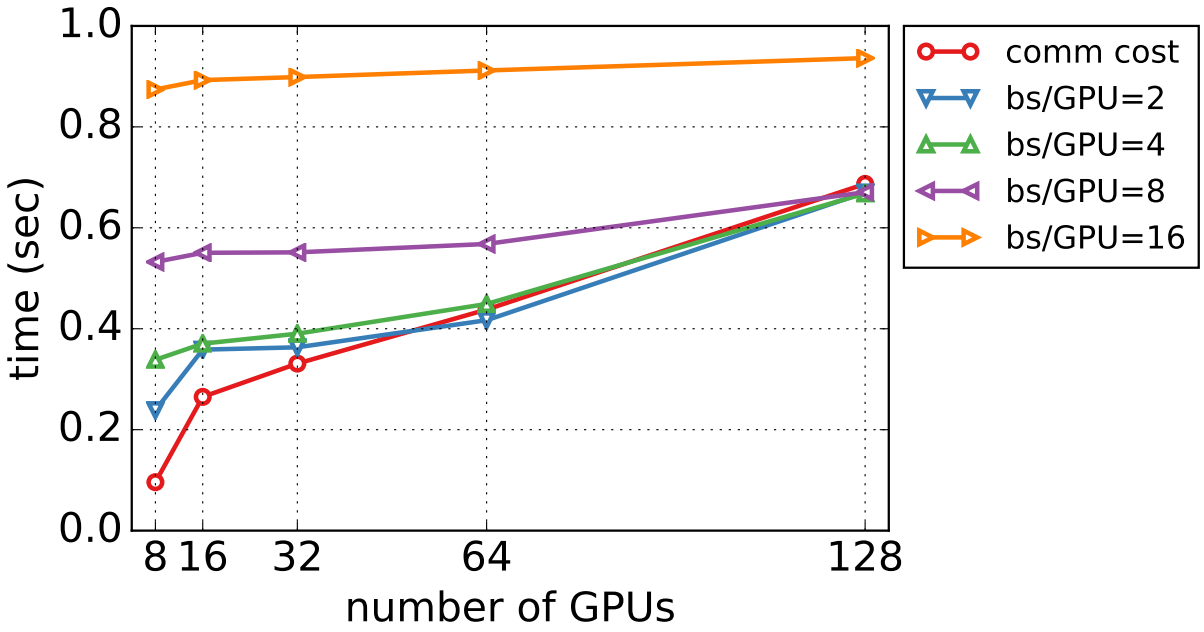


Figure 4.11: The communication cost and total cost of one SGD iteration. Experiments are performed on multiple machines (1,2,4,8,16), and the number of GPUs per machine is fixed to be 8.

We tune the learning rate as follows: we start the training process with a learning rate 0.1; then we reduce the learning rate by a factor of 10 at epoch 30, 60, and 90, respectively; and we further stop data augmentation at epoch 100. As shown in Figure 4.12, for the baseline, we obtained 77.8% top-1 validation accuracy at convergence (at epoch 110), which matches the reported result of 77% top-1 validation accuracy (at epoch 90) in [65]. We then increase the number of GPUs from 8 to 80, and thus the aggregate batch size increases from 256 to 2,560 (2K). We change the initial learning rate from 0.1 to 0.5. For 160 GPUs and the aggregate batch size 5,120 (5K), we further increase the initial learning rate to 1 and defer the learning rate reduction from epoch 30 to epoch 50. Intuitively, these heuristics introduce more "variance" to the training process.

In Figure 4.12, we plot the top-1 validation accuracy versus the number of SGD epochs. Observe that with our carefully tuned learning rate, the batch size does not significantly affect the algorithm convergence. In particular, the convergence for batch size 2K is very close to that of the baseline; for batch size 5K, despite the unstable convergence in the beginning of the training process, it converges to the same final accuracy of the baseline as the training progresses.

4.6 Discussions

MXNet is a efficient and flexible deep learning library. To reduce communication cost, it features a simplified communication interface supported by a two-level parameter server architecture.

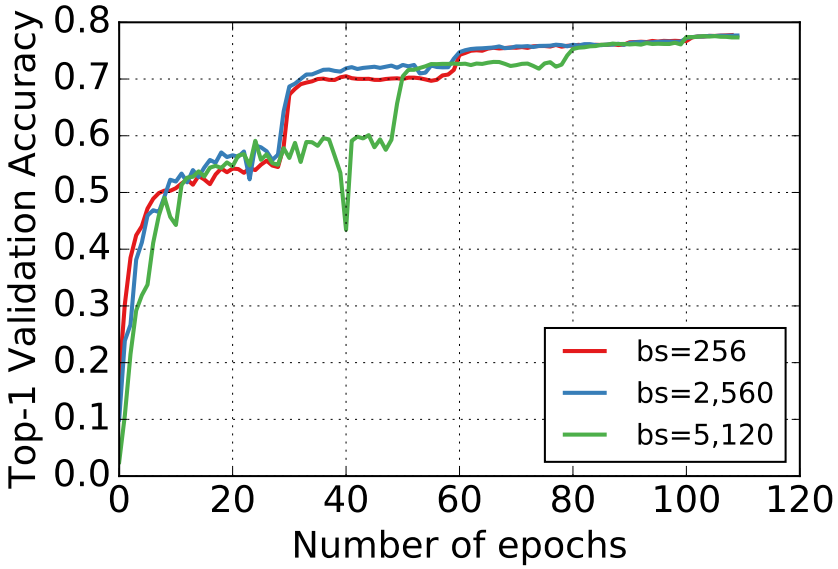


Figure 4.12: Top-1 validation accuracy versus epoch for Resnet-152 on Imagenet dataset. Each GPU uses batch size 32 and synchronized SGD is used.

We have demonstrated encouraging experiment results of MXNet in a large scale distributed computing environment.

MXNet is an ongoing project. We started it in 2015 based on the two projects of CXXNet [46] and Minerva [36]. MXNet inherits the Numpy-like interface and features such as lazy evaluation, and develops a new front-end with a mixed interface and multi-language support, as well as a new back-end with memory optimization and distributed computing. To move forward, for the front-end interface, we are making it easier to use, as easy-to-use is of great importance for users new to a framework in the fast growing deep learning community today. For the back-end system, we will continue to explore the technologies of compiler and database, and use them in MXNet to develop a more efficient back-end implementation. Moreover, with a modularized system design where the system components can be decoupled, our MXNet will continue to evolve and promptly adapt itself to the developing neural network models and hardware devices.

Part II

Algorithm

Chapter 5

Preliminaries on Optimization Methods for Machine Learning

In this chapter, we provide some background information on the interplay between optimization methods and machine learning, which will be helpful for our discussion in the second part of this thesis. For a thorough overview of modern optimization methods, one is referred to the textbook [17, 22, 106]. The books [19, 126] contain more detailed discussions on the topics of machine learning and large scale optimization.

In the rest of this chapter, we first present the generic optimization formulation that is central to the model training of various machine learning problems, and overview a few standard techniques for solving the optimization. Then we introduce some terminologies in convergence analysis for iterative methods. At last, we discuss distributed optimization, which is vital for large scale optimization problems.

5.1 Optimization Methods

For a lot of machine learning applications, the model training problem can be formulated into the following optimization problem:

$$\underset{w\in\Omega}{\text{minimize}}\ f(w) = \frac{1}{n}\sum_{i=1}^{n} f_i(w), \tag{5.1}$$

where w denotes the model parameter and Ω denotes the parameter space.The function $f_i(\cdot)$ is the cost function evaluated with i-th example in the training data, describing how well the model w fits the particular example in the training data.

For general cost functions, there is usually no closed form solution to the above optimization problem, and we need to resort to numerical methods. One class of commonly used numerical methods is called iterative gradient descent method. As can be inferred by its name, this method uses the gradient information of the objective function $f(\cdot)$ at different points to iteratively update the model parameter w. It usually starts with an initial point $w_0 \in \Omega$ at $t = 0$, and then updates

the model parameter according to the following parameter update rule at iteration t:

$$w_{t+1} = \operatorname*{proj}_{\Omega} \left[w_t - H_t \sum_{i \in I_t} \partial f_i(w_t) \right] \quad \text{for } I_t \subseteq \{1, \ldots, n\} \tag{5.2}$$

where ∂f_i is the partial gradient of f_i with respect to the model parameter w, the constant H_t is a scaling matrix or scalar, and the set I_t is a subset of example indices which are processed at iteration t. Different choices of H_t and I_t lead to different optimization methods. For instance, in the standard gradient descent method, we set I_t to be all the training data, and set H_t to be a constant scalar. The iterations terminate if a stopping criteria is reached, for example when the progress on the objective function is below certain threshold.

Next, we discuss several commonly used variants of the iterative gradient descent method.

Batch vesus Stochastic In batch update, all the examples in the training data are used to compute the gradient in each iteration, namely $I_t = \{1, \ldots, n\}$. For stochastic gradient update, in each iteration one usually only samples a few random examples and compute the gradient information of the cost function on these examples. Compared to batch update, the gradient information used in stochastic methods is noisier due to the fact that only a few examples are sampled. However, for the same reason, each iteration of stochastic update is much faster as we only need to compute the gradients on the samples. In a fixed amount of time more iterations can be run for stochastic update.

Jacobi versus Gauss-Seidel When the model size is large, or equivalently the number of coordinates of the parameter w is large, it may not be necessary to compute the gradient with respect to every coordinate and update every coordinate in each iteration. In Jacobi method, all the coordinates of w are updated in each iteration. In Gauss-Seidel method, a subset of coordinates in w is selected in each iteration. The gradient information of the objective function with respect to this subset of coordinates are obtained and used to update these coordinate. Note that this may be done much faster than computing the gradients with respect to all the coordinates especially when the model size is large. One special case of Gauss-Seidel method is called coordinate descent, where in each iteration only one coordinate of the model parameter is selected and updated.

Proximal Gradient Method The proximal gradient method [34, 108] is often used to handle the non-differential objective functions. Assume that the objective function $f(\cdot)$ can be decomposed into two parts $f(w) = \ell(w) + h(w)$ where $\ell(\cdot)$ is differentiable and $h(\cdot)$ is not. Proximal gradient method updates the parameter w in two steps: a forward step that performs steepest gradient descent only on $\ell(\cdot)$, and a backward step that carries out projection using $h(\cdot)$. More specifically, the t-th iteration with a learning rate γ_t can be written as:

$$w_{t+1} = \text{Prox}_{\gamma_t}^{U} \left[w_t - \gamma_t \partial \ell(w_t) \right], \tag{5.3}$$

where the generalized proximal operator for $h(\cdot)$ is defined to be:

$$\text{Prox}_{\gamma}^{U}(x) := \operatorname*{argmin}_{y \in \Omega} \; h(y) + \frac{1}{2\gamma} \|x - y\|_U^2 . \tag{5.4}$$

The Mahalanobis norm $\|x\|_U := x^\top U x$ is taken with respect to a predefined positive semidefinite matrix U. A sophisticated choice of U can accelerate the convergence of the iterative algorithm.

Approximate Second Order Methods There are situations when the standard gradient descent may fail to make sufficient progress on the objective function over iterations. For example, if the condition number of the Hessian matrix, evaluated at the points along the path of iteration, is too large, the gradients, which points to the maximum direction of change in the objective function, are not well-aligned with the directions pointing to the minimum solution, and it would take too many iterations for gradient descent to converge to that minimum solution. In this case, second-order information of the objective function can be used to improve the convergence. In the classic Newton method, the inverse of the Hessian matrix is used as the scaling matrix, namely setting the scaling matrix $H_t = (\sum_{i \in I_t} \partial^2 f_i(w_t))^{-1}$.

However, for large scale models, computing such a scaling matrix requires the inversion of a giant matrix in each iteration, which is often too expensive in practice. There are different approaches to approximate the Hessian matrix and the scaling matrix. One straightforward approximation is to only compute the diagonal entries of the Hessian matrix while setting the off-diagonal entries to zero. Another widely used approach is to approximate the Hessian matrix using the gradients. For example L-BFGS [89] forms an approximation of the Hession matrix using the gradients over the past iterations.

5.2 Convergence Analysis

For general objective functions, there is usually no guarantee that iterative gradient descent methods will converge to the minimum of the optimization problem in (5.1). The best hope is that the sequence of model parameter $\{w_0, w_1, \ldots\}$ converges to a "good enough" solution. We list three types of "good" solutions characterized by the value of objective function and gradient information at the solution $\hat{w}$.

1. Global minimum: for any $w \in \Omega$, it is true that $f(w) \geq f(\hat{w})$.
2. Local minimum: given a fixed radius $\epsilon > 0$, for any $w \in \Omega$ and in the range $\|w - \hat{w}\| < \epsilon$, it is true that $f(w) \geq f(\hat{w})$.
3. Stationary point: the gradient namely $\partial f(\hat{w}) = 0$.

Note that in the special case where the objective function is convex, the above characterizations can be shown to be equivalent.

Convergence analysis studies the conditions under which an iterative algorithm converges to a good solution. For gradient based iterative algorithm with general objective functions, usually we aim to prove the convergence to a stationary point.

We are interested the convergence rate of iterative algorithms, defined as the speed at which the convergent sequence approaches its limit. It is often used as a performance measure to compare different algorithms. Consider a strongly convex objective function for which the global optimal solution is denoted by w^*. For batch gradient descent, it can be shown that there exists a constant $\rho \in (0, 1)$ such that at iteration t, we can bound

$$f(w_t) - f(w^*) \leq O(\rho^t).$$

In other words, to find a solution $\hat{w}$ such that the objective function evaluated at $\hat{w}$ is at most $(f(w^*) + \epsilon)$, the number of iterations required is in the order of $O(\log(1/\epsilon))$. In particular, we call this linear convergence rate. If using stochastic gradient descent, the convergence speed is also stochastic for the randomized algorithm. One can show that in expectation,

$$\mathbb{E}[f(w_t)] - f(w^*) \leq O(1/t).$$

Namely, for stochastic gradient descent to achieve the same value of objective function, the number of iterated needed is in the order of $O(1/\epsilon)$. However, recall that one iteration in SGD is often much cheaper than an iteration in batch gradient descent. The performance measure of convergence rate should be used with caution. In practice, we also need to compare the actual runtime of two iterative algorithms.

5.3 Distributed Optimization

Today, many machine learning problems are large scale. When the size of the datasets and / or the model size is too large, even one iteration of gradient descent method cannot be computed in reasonable time by a single machine. Distributed optimization leverages multiple machines to process the computation in a parallelized or decentralized fashion. There are two major degrees of freedom for designing a distributed optimization method: first, how to partition the computational workloads and assign them to different machines; second, how to communicate between machines in order to synchronize the computation results to ensure the convergence of the algorithm.

5.3.1 Data Parallelism versus Model Parallelism

There are two common approaches to partition the computational workloads: data parallelism and model parallelism.

In data parallelism, the examples in the training data are partitioned and assigned to different machines. Take batch gradient descent method as an example. Each machine only computes the gradient of the cost function evaluated on the examples assigned to it. The gradient computation can be done in parallel, and these local gradients are then merged to obtain the gradient of objective function.

Model parallelism partitions the coordinates of the model parameter w and assign them to different machines, while each machine has access to the entire training data. In each iteration of gradient descent, each machine computes the gradients of the objective function with respect to the coordinates assigned to it.

Data parallelism is mostly used when the size of training data is too large to fit into a single machine, while model parallelism is often used when the number of model parameters is large.

5.3.2 Synchronous Update versus Asynchronous Update

While multiple machines doing local computation in parallel, if the original optimization problem is not perfectly decoupled, the machines need to communicate with each other about the

results of local computation along the iterations. For example, in data parallelism, gradients of the local assigned examples are computed by every machine in each iteration, and the updated model parameter in the last iteration is needed for to compute the local gradients.

Synchronous update refers to that in each iteration, the model is updated after the local computation at every machine is completed. The same convergence result as that of single machine is guaranteed. However, it potentially incurs a large latency for each iteration, especially when there are a few machines that are particularly slow, and in this case the computing power of the machines may not be fully utilized.

Asynchronous update allows different machines to use "staled" information of the computation results from other machines for their local computation in the current iteration, without waiting for the all the machines to complete local computation. For example, in model parallelism, one machine can compute its local gradients using an outdated model without waiting for the most recent update from every other machine. Asynchronous update can greatly reduce the synchronization cost. However, if not carefully designed, it may also slow down the convergence of the algorithm.

Chapter 6

DBPG: Delayed Block Proximal Gradient Method

6.1 Introduction

The objective function in the optimization for many model training problems can be decomposed into two parts—loss and regularization. The loss measures how well the model fits the data while the regularization penalizes too complex models. In this chapter, we consider the following objective function which can be decomposed into two parts: the loss function $\ell(\cdot)$ and the regularization term $h(\cdot)$:

$$\underset{w \in \Omega}{\text{minimize}} \; f(w) = \ell(w) + h(w) \text{ and } w \in \mathbb{R}^p. \tag{6.1}$$

We assume that the loss function $\ell : \Omega \to \mathbb{R}$ is continuously differentiable but *not* necessarily convex, and that the regularizer $h : \Omega \to \mathbb{R}$ is convex, left side continuous, block separable, but possibly *non-smooth*.

We propose a proximal gradient method [108] based algorithm, called Delayed Block Proximal Gradient (DBPG), to solve the above optimization problem. Our algorithm is significantly more efficient than the existing algorithms especially in the regime where the dimension of w is high (millions or billions coordinates) and the data is sparse (existing many zero entries). Its success is contributed by the following facts:

- Taking advantage of the fact that (block) Gauss-Seidel updates are more efficient on sparse data [116, 151], our algorithm only updates a subset (block) of coordinates in each iteration.
- The proximal operator uses coordinate-specific learning rates to adapt the convergence progress to the sparsity pattern inherent in the data.
- In the setting of distributed computing, we adopt asynchronous communication. To reduce network traffic, each worker node only maintains partially consistent model parameters, and the coordinates that would change the associated model weights are communicated.

We demonstrate the efficiency of our algorithm by applying it to two challenging problems: first, a non-smooth ℓ_1-regularized logistic regression on sparse text datasets with over 100 billion

Algorithm 5 Delayed Block Proximal Gradient Method (DBPG) Solving (6.1)

Scheduler:

1: Partition examples into m parts $\bigcup_{k=1}^{m} I_k = \{1, \ldots, n\}$
2: Partition parameters into b blocks $\bigcup_{k=1}^{b} B_k = \{1, \ldots, p\}$
3: **for** iteration $t = 1$ to T **do**
4: Randomly pick a block b_{t_i} and issue the task to workers
5: **end for**

Worker k at iteration t

1: Wait until all iterations before $t - \tau$ are finished
2: Compute the gradient $g_t^{(k)} = \sum_{i \in I_k} \partial \ell_t(w_t^{(k)})$ and coordinate-specific learning rates $u_t^{(k)}$ on block b_{t_i}
3: Push $g_t^{(k)}$ and $u_t^{(k)}$ to servers with user-defined filters, e.g., the random skip or the KKT filter
4: Pull $w_{t+1}^{(k)}$ from servers with user-defined filters, e.g., the significantly modified filter

Servers at iteration t

1: Aggregate g_t and u_t
2: Update w_{t+1} by solving the the generalized proximal operator (5.4) defined in Section 5.1 with $U = \text{diag}(u_t)$.

examples and features; second, a non-convex and non-smooth ICA reconstruction problem [80], where the goal is to extract billions of sparse features from dense image data.

Related work There is growing interest in asynchronous iterative algorithms for optimization. Shotgun [23] performs parallel coordinate descent for solving ℓ_1 optimization problems. Other methods [7, 79, 138, 157] partition examples and distribute them over several machines and update the model parameters in a data parallel fashion. Lock-free variants were proposed in Hogwild! [113]. Mixed variants which partition data and parameters into non-overlapping components were introduced in [137], albeit at the price of having to move or replicate data on several machines. Lastly, the NIPS framework [125] discusses general non-convex approximate proximal methods.

The proposed algorithm differs from the existing approaches mainly in two aspects. First, we focus on solving large scale problems. Given the sheer size of training data and the limited network bandwidth, neither the shared memory approach of Shotgun and Hogwild! nor the approach of moving the entire data during the training process is applicable. We carry out experiments at a scale that is of many orders of magnitude larger. Second, we aim at solving general non-convex and non-smooth objective functions. We obtain results about the convergence rate with weaker assumptions than that in [125].

6.2 Delayed Block Proximal Gradient Method

6.2.1 Proposed Algorithm

Algorithm 5 gives a sketch of the proposed algorithm, named Delayed Block Proximal Gradient (DBPG). It takes advantage of the opportunities offered by our Parameter Server framework (see Chapter 3) to handle high-dimensional sparse data. Next, we highlight how our algorithm substantially differs from the standard methods including gradient descent and distributed gradient descent.

1. Only a subset of parameters is updated in each iteration.
2. The worker nodes compute both local gradients and coordinate-specific learning rates.
3. We use a bounded-delay model for the asynchronous iterations. Given the maximal delay τ, the iteration t can be started without waiting for any previous iteration initiated in the time frame $[t-\tau, t]$ to be completed. Here, τ is a trade-off between data consistency and system performance. For $\tau = 0$, it corresponds to the standard sequential updates, while for a larger τ, it allows more iterations to run in parallel, which potentially improves the system efficiency.
4. To reduce communication overhead, we apply user-defined filters to suppress transmission of the updates whose effect on the model are likely to be negligible.

6.2.2 Convergence Analysis

We introduce a few assumptions to simplify the convergence analysis for the proposed algorithm. Under the assumption that the loss function is additive, let $\ell_{I_k} = \sum_{i \in I_k} \ell_i$ denote the loss function associated with the subset of training data assigned to worker node k. Consider the subset of parameters B_t chosen at iteration t. Assumption 1 below ensures that the gradients of ℓ_{I_k} with respect to a subset of parameters are Lipschitz, and the amount of "cross-talk" to other subsets of parameters are also bounded.

Assumption 1 (Block Lipschitz Continuity) *There exists positive constants $L_{var,k}$ and $L_{cov,k}$ such that for any iteration t and all $x, y \in \Omega$ with $x_i = y_i$ for any $i \notin B_t$, we have*

$$\|\partial_{B_t} \ell_{I_k}(x) - \partial_{B_t} \ell_{I_k}(y)\| \le L_{var,k} \|x - y\|, \quad \textit{for } 1 \le k \le m; \tag{6.2a}$$

$$\|\partial_{B_s} \ell_{I_k}(x) - \partial_{B_t} \ell_{I_k}(y)\| \le L_{cov,k} \|x - y\|, \quad \textit{for } 1 \le k \le m, t < s \le t + \tau. \tag{6.2b}$$

where $\partial_B \ell(x)$ is the block B of $\partial \ell(x)$. Further define $L_{var} := \sum_{k=1}^m L_{var,k}$ and $L_{cov} := \sum_{k=1}^m L_{cov,k}$.

Theorem 2 shows that the proposed algorithm converges to a stationary point under the relaxed consistency model, provided that a suitable learning rate is chosen. Note that for general objective functions there is no guarantee of global optimality.

Theorem 2 *Assume that in Algorithm 5, the updates are performed with a τ bounded-delay model, and assume that Assumption 1 holds. Also assume that we apply a random skip filter on pushing gradients and a significantly-modified filter on pulling weights with threshold $\mathcal{O}(t^{-1})$.*

Let M_t denote the minimal coordinate-specific learning rate at time t. Algorithm 5 converges to a stationary point in expectation if the learning rate γ_t satisfies

$$\gamma_t < \frac{M_t}{L_{var} + \tau L_{cov} + \epsilon}, \quad \textit{for all } t > 0. \tag{6.3}$$

The detailed proof of the above theorem is given in Section 6.4. Intuitively, the difference between $w_{t-\tau}$ and w_t will be small near a stationary point. As a consequence of the Lipschitz assumption, the changes in the gradients will also vanish. The inexact gradient obtained by delayed and inexact model, therefore, is likely to be a good approximation of the true gradient, so that the standard convergence results of proximal gradient methods can be applied.

Note that, when the maximum delay τ increases, it may not be necessary to decrease the learning rate in order to guarantee convergence. Instead, one can carefully choose a block partition of the model parameters and ordering. For example, if features in a block are less correlated then L_{var} decreases. If the block is less correlated to the previous blocks, then L_{cov} decreases. This was also exploited in [23, 113].

We apply two filters to reduce the data communication overhead. One filter is to randomly set some gradient entries to 0, which is also kwown as "dropout" in the deep learning literature [129]. The second filter is to skip pulling the entries that have not been updated for more than $\mathcal{O}(t^{-1})$ time slots. Surprisingly, out experiments show that applying these two filters in the iterations do not slow down the convergence.

6.3 Experiments

In this section, we demonstrate how the proposed algorithm DBPG can be used to solve challenging machine learning problems. We first consider the non-smooth sparse logistic regression, and then we look into the more challenging problem of Reconstruction ICA which objective function is both non-convex and non-smooth.

6.3.1 Sparse Logistic Regression

The objective function of sparse logistic regression is

$$\underset{w \in \mathbb{R}^p}{\text{minimize}} \sum_{i=1}^{n} \log(1 + \exp(-y_i \langle x_i, w \rangle)) + \lambda \|w\|_1 \tag{6.4}$$

where x_i is a p-dimensional vector and $y_i \in \{+1, -1\}$. Note that the ℓ_1-regularizer is non-smooth at zero.

Implementation

We implement Algorithm 5 on our proposed Parameter Server framework (see Chapter 3). In particular, we apply upper bounds of the diagonal entries of the Hessian for the coordinate-specific learning [154]. The learning rate is chosen from the grid of $[0.2, 0.3, \ldots, 0.9, 1]$, and we

search for the largest learning rate that ensures algorithm convergence. Moreover, we design a Karush-Kuhn-Tucker (KKT) filter which works to skip updating inactive coordinates. Its principle is analogous to that of the active-set selection strategies of SVM optimization [71, 96]. In particular, if $w_i = 0$ for the i-th coordinate, let g_i denote the gradient with respect to the i-th coordinate at the current iteration. According to the optimality condition of the proximal operator, also known as the soft-shrinkage operator, the model parameter w_i remains 0 as long as the gradient $|g_i| \leq \lambda$, and thus the worker does not need to send the gradient g_i to the server. We use an outdated value $\hat{g}_i$ to approximate g_i, and the i-th coordinate is skipped if $|\hat{g}_i| \leq \lambda - \delta$, where $\delta \in [0, \lambda]$ controls how aggressive the filtering is.

Performance on Single Machine

To the best of our knowledge, no open source system can handle sparse logistic regression to the scale in our work. Graphlab provides only a multi-threaded, single machine implementation; and Mlbase, Petuum and REEF do not support sparse logistic regression (as confirmed with the authors in 4/2014). Therefore, we could only compare our implementation to other solvers in their multicore setting if available, and limiting to a relatively small dataset.

More specifically, we compare our Parameter Server implementation of Algorithm 5 to Shotgun [23][1] on a single machine with 32 threads/workers. The performance of CDN [51] (single thread Shotgun) is also reported for reference. Figure 6.1 shows how the objective values decreases over time. All three implementations obtain similar objective values after 50 data passes, however, our algorithm is 4 times faster.

We gain the speedup with clever data partitioning strategies. In each iteration, each thread of Shotgun only processes a single coordinate. The irregular pattern of non-zero entries makes it hard to perform load balancing. On the most high-dimensional dataset CTRa, Shotgun is even slower than the single thread implementation. On the other hand, our Parameter Server implementation uses multi-thread linear algebra operators on much larger blocks of training data, which coarse-grained parallelization leads to better load balancing and overall speedup.

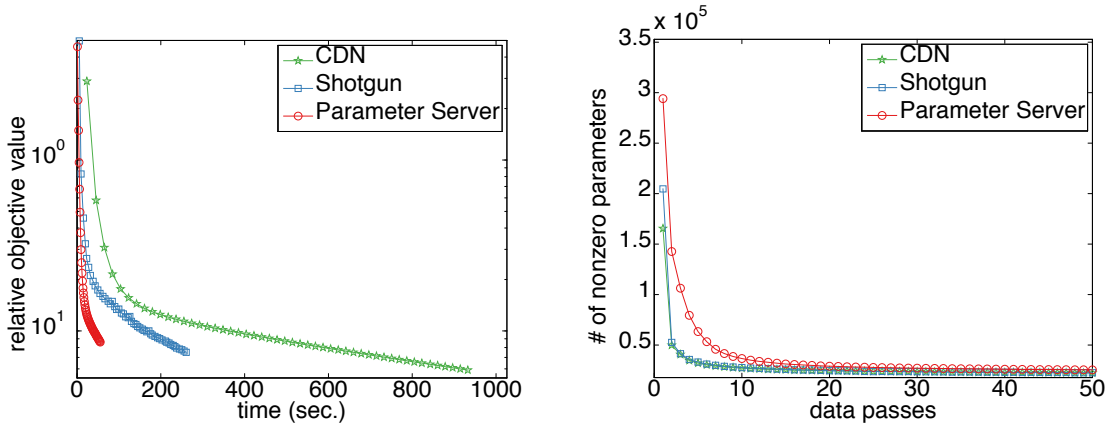
Also note that the proposed algorithm produces less sparser models compared to other algorithms during the first few iterations, which might be due to our usage of large block size for features. However, when close to convergence, the model produced by our algorithm has sparsity level comparable to other algorithms.

Performance on Multiple Machines

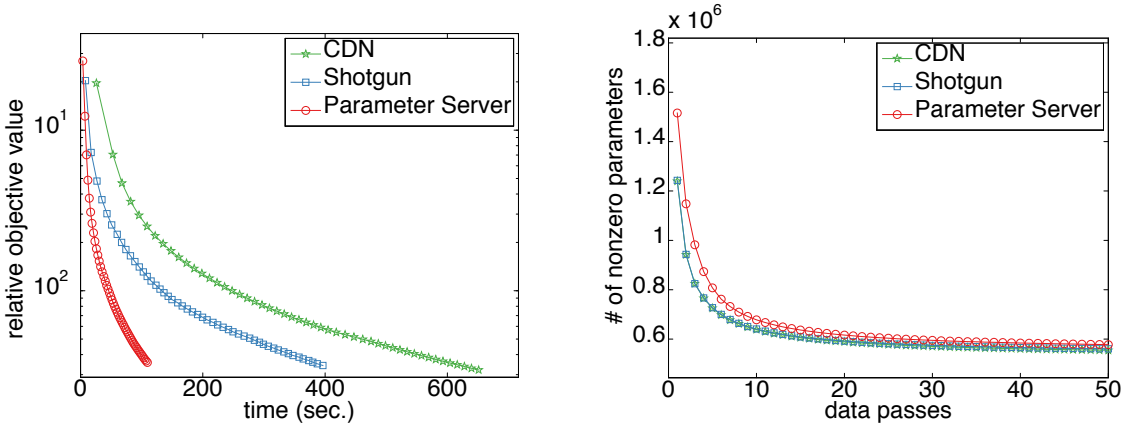
We compare our Parameter Server implementation of the proposed algorithm with two special-purpose second generation parameter servers, named system A and B, developed by a leading Internet company. Both systems adopt the sequential consistency model, while system A uses a variant of L-BFGS [11, 89] and system B runs an algorithm that is similar to ours but does not allow any data delay (i.e. synchronous update). We again run the implementations to reach the same convergence criteria.

Figure 6.2 shows that System B outperforms system A due to its better algorithm. Our parameter server implementation is 2 times faster than System B for essentially the same algorithm.

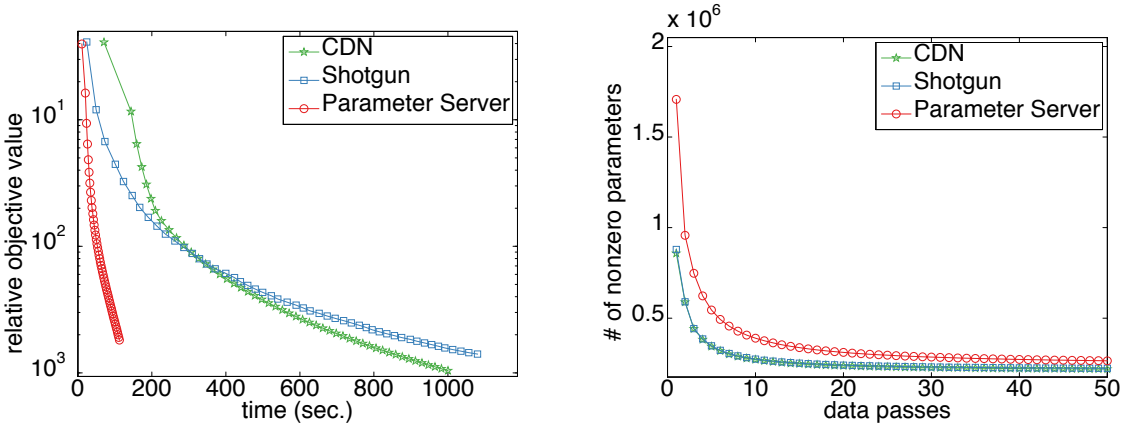
[1] www.select.cs.cmu.edu/projects/shotgun/



(a) Dataset: URL



(b) Dataset: KDD14



(c) Dataset: 4 million examples sampled from CTRb

Figure 6.1: Comparison between Parameter Server implementation of Algorithm 5 and Shotgun and CDN implementation.

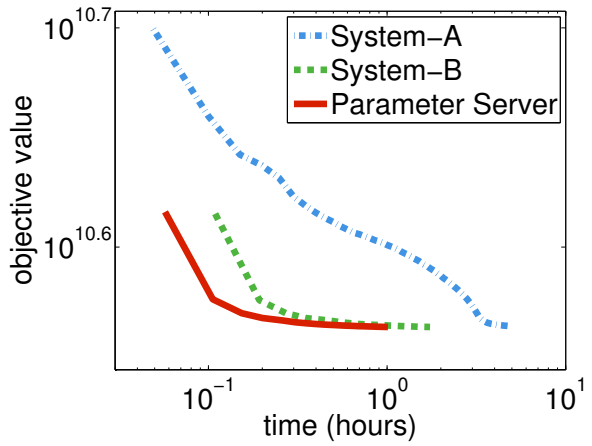


Figure 6.2: Convergence of sparse logistic regression on 636TB CTRb.

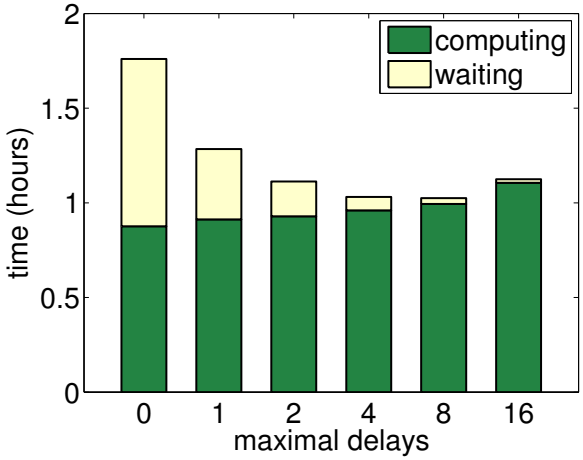


Figure 6.3: Time to reach the same convergence criteria under various allowed delays.

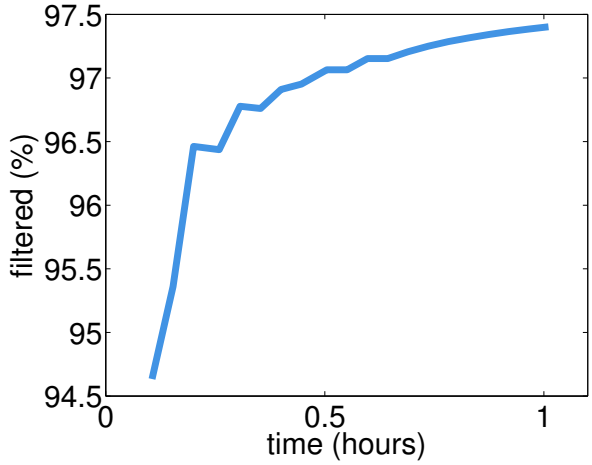


Figure 6.4: Percentage of coordinates skipped when using the KKT filters.

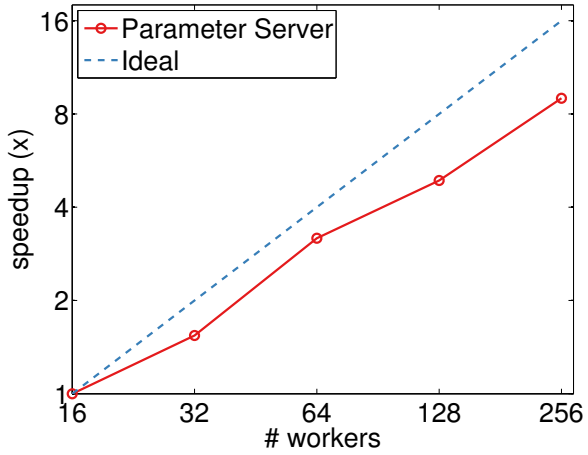


Figure 6.5: Speedup of Parameter Server when increasing the number of workers with a fixed number of servers. The dataset is 340 million examples sampled from CTRb.

Figure 6.3 shows that increasing the allowed delays significantly decreases the waiting time at the cost of slightly slower convergence. The best trade-off is 8-delay with a 1.6x speedup compared to the sequential consistency model.

Figure 6.4 demonstrates the effectiveness of KKT filter. Observe that in less than 10 minutes after the training process starts, our algorithm makes more than 95% coordinates inactive. It suggests that using the filter can potentially improve the performance, as we do not need to compute and communicate the gradients of those inactive coordinates.

Figure 6.5 demonstrate the scalability of our algorithm. It shows a 9x speedup when the

number of worker nodes increases from 16 to 256.

6.3.2 Reconstruction ICA

Reconstruction Independent Component Analysis (RICA) aims to find a sparse representation of the raw dataset. Compared to the standard Independent Component Analysis (ICA), RICA allows an overcomplete solution [80]. The objective function of RICA has a non-convex loss function and convex but non-smooth penalty function as below.

$$\underset{W\in\mathbb{R}^{\ell\times p}}{\text{minimize}} \sum_{i=1}^{n} \frac{1}{2}\left\|WW^\top x_i - x_i\right\|_2^2 + \lambda\left\|Wx_i\right\|_1. \tag{6.5}$$

We denote the observations $\{x_i\}_{i=1}^n \in \mathbb{R}^p$ by $X = (x_1, \ldots, x_n)^\top \in \mathbb{R}^{n\times p}$ the data matrix. The gradient of the loss function is given by:

$$\partial\ell(W) = W\left((W^\top W - I)X^\top X + X^\top X(W^\top W - I)\right). \tag{6.6}$$

This can be seen by rewriting the objective function $\ell(W)$ using trace identities:

$$\begin{aligned}
\ell(W) &= \frac{1}{2}\left\|XW^\top W - X\right\|_F^2 \\
&= \frac{1}{2}\operatorname{tr}\left(W^\top WX^\top XW^\top W - 2X^\top XW^\top W + X^\top X\right) && (\|A\|_F^2 = \operatorname{tr}(A^\top A)) \\
&= \frac{1}{2}\operatorname{tr}\left(W^\top WX^\top XW^\top W\right) - \operatorname{tr} X^\top XW^\top W + \frac{1}{2}\operatorname{tr} X^\top X && (\operatorname{tr}(A+B) = \operatorname{tr}(A) + \operatorname{tr}(B)) \\
&= \frac{1}{2}\operatorname{tr}\left(XW^\top WW^\top WX^\top\right) - \operatorname{tr}\left(WX^\top XW^\top\right) + \frac{1}{2}\operatorname{tr}\left(X^\top X\right) && (\operatorname{tr}(AB) = \operatorname{tr}(BA))
\end{aligned}$$

Applying (112) and (100) of [109] directly we obtain (6.6).

Implementation

In this problem, invoking the proximal operator in our algorithm is nontrivial, as $\|WX\|_1$ is not separable but only block separable. Therefore, we need to solve each of the following n independent optimization problems simultaneously using ADMM [21]:

$$\underset{u_i}{\text{minimize}} \ \frac{1}{2\gamma}\left\|u_i - z_i\right\|_{H_i}^2 + \lambda\left\|Xu_i\right\|_1, \quad \text{for } i = 1\ldots n, \tag{6.7}$$

where we set $z_i = w_i - \gamma H_i^{-1}\partial_i\ell(W)$, and $w_i \in \mathbb{R}^p$ denotes the i-th row of the parameter W. Here γ is the learning rate and $H_i \in \mathbb{R}^{d\times d}$ are scaling matrices of the metric space. For notational convenience we drop the subscript i when it is clear in the context. Following [42], we choose the scaling matrices at iteration t to be:

$$H_t^2 = H_{t-1}^2 + \operatorname{diag}(w_t - w_{t-1})^2 \text{ for } t \geq 0 \quad \text{and} \quad H_0 = \mathbf{1},$$

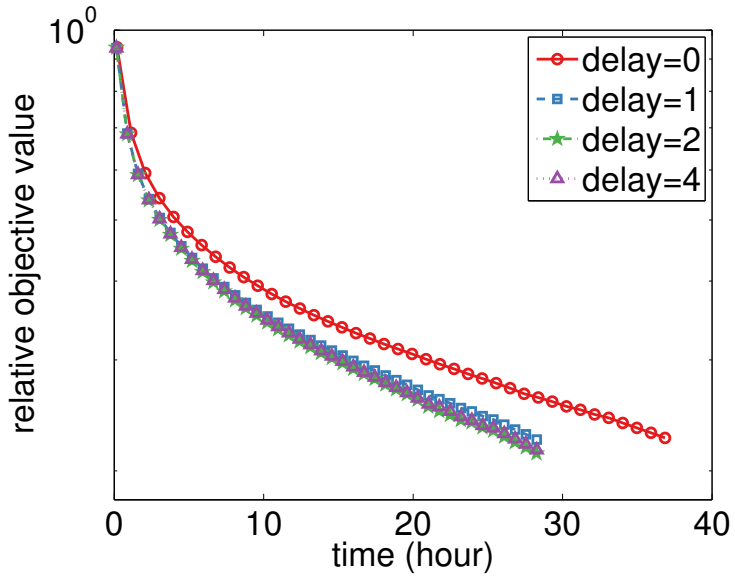


Figure 6.6: Convergence of RICA on dataset ImageNet with different delays.

which can be computed locally. With the auxiliary variable $y := Xu$, the augmented Lagrangian is given by:

$$L(u, y, \mu) = \frac{1}{2\gamma} \|u - z\|_H^2 + \lambda \|y\|_1 + \langle \mu, Xu - y \rangle + \frac{1}{2\theta} \|Xu - y\|^2 . \tag{6.8}$$

Let $S_\lambda(\cdot)$ be the soft-thresholding function. We obtain the corresponding update rules as:

$$u \leftarrow \left(\gamma^{-1} H + \theta^{-1} X^\top X\right)^{-1} \left(\gamma^{-1} H z + X^\top \left(\theta^{-1} y - \mu\right)\right) \tag{6.9a}$$

$$y \leftarrow S_\lambda \left(\theta^{-1} X u + \mu\right) \tag{6.9b}$$

$$\mu \leftarrow \mu + \theta^{-1} \left(Xu - y\right), \tag{6.9c}$$

Note that in our Parameter Server implementation, each worker node can update its parameters independently if it has the training data X and has access to the matrix $W^\top W$, which dimension is typically much smaller than that of W. Therefore, we apply parameter partition for RICA, where the server maintains $W^\top W$, and each worker has a copy of X and only a subset of rows of W.

Since most computations are dense matrix operations, we implement the proposed algorithm with GPUs using CUBLAS, which uses all the computational units within the GPU by default. We configure one worker node on each GPU card. We run the experiments on cluster CampusB where each machine is equipped with an Nvidia Tesla K20. To obtain the training set, we randomly sub-sample 100,000 images from ImageNet and resize them to 100×100 pixels.

Performance

Figure 6.6 shows how delay affects the algorithm convergence speed. Similar to the simpler problem of ℓ_1-regularized logistic regression, we can observe the improvement caused by asynchronous updating. However, increasing the delay beyond 1 slightly affects the time of each

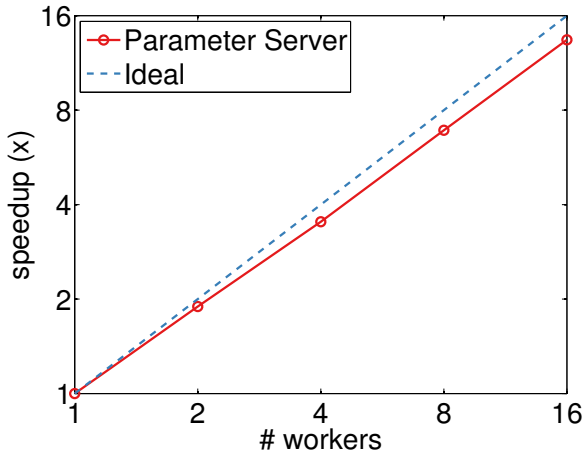


Figure 6.7: Speedup of Parameter Server when increasing the number of workers from 1 to 16 for RICA.

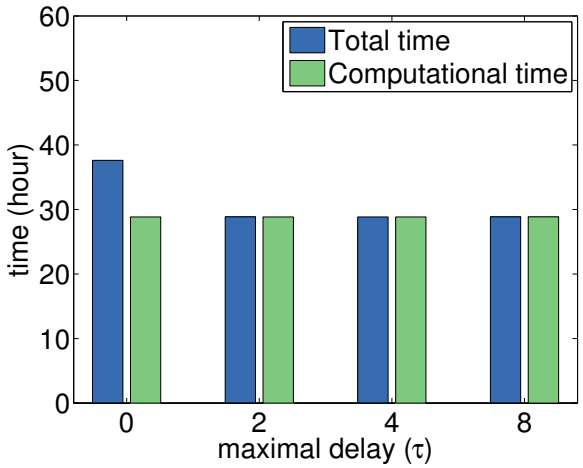


Figure 6.8: Comparison between computation time and total time for RICA.

iteration and the convergence rate. This is because the update delay observed of RICA is typically 1.

In Figure 6.7, we see a 13.5x acceleration for RICA when increasing the number of workers by 16 times. The larger speedup of RICA compared to ℓ_1-regularized logistic regression is because that RICA mainly consists of dense matrix operations, which are easier to balance than sparse matrices operations. This point is illustrated by Figure 6.8, where we observe that even for synchronous updating (0 delay), over 80% of the total cost of RICA is contributed by the actual computation.

6.4 Proof of Theorem 2

We first prove several technical lemmas towards the proof of Theorem 2.

Let $B \subseteq \{1, \dots p\}$ denote a subset of coordinates and let $x_B \in \mathbb{R}^p$ denote the vector obtained by setting the entries of x, which are not in block B, to 0. We first show that under Assumption 1 the objective is well behaved under subspace shifts.

Lemma 3 *Assume block B is chosen at time t, then the following holds under Assumption 1 for any ℓ_{I_k} and for any time t, for any $x, y \in \mathbb{R}^p$*

$$\ell_{I_k}(x + y_B) \leq \ell_{I_k}(x) + \langle \partial \ell_{I_k}(x), y_B \rangle + \frac{L_{var,k}}{2} \|y_B\|^2 , \tag{6.10}$$

Proof. By the mean value theorem it follows that

$$\ell_{I_k}(x + y_B) = \ell_{I_k}(x) + \langle \partial \ell_{I_k}(x + \xi y_B), y_B \rangle \text{ for some } \xi \in [0, 1]. \tag{6.11}$$

Using the Lipschitz property of Assumption 1 it follows that the gradient at $x + \xi y_B$ can be bounded via $|\partial \ell_{I_k}(x + \xi y_B) - \partial \ell_{I_k}(x)| \leq L_{\text{var},i} \xi \|y_B\|$. Combining this with $\xi \leq 1$ proves the claim. ■

Next we prove that for block separable regularizers the solutions also satisfy an appropriate decomposition property:

Lemma 4 *Assume that h is block separable and $0 \in \partial h(0)$ and that U is diagonal. For any x and for $\gamma > 0$ we denote the solutions of the proximal operator to the full vector, and that of only a subset, by $z = \text{Prox}_\gamma^U(x)$ and $z_B = \text{Prox}_\gamma^U(x_B)$, respectively. For any block b the following holds:*

$$U\left(x_B - z_B\right) \in \gamma \partial h(z_B) \tag{6.12}$$

Proof. Since $0 \in \partial h(0)$ it follows that $\text{Prox}_\gamma(0) = 0$. Further since h is block separable, the proximity function $h(y) + \frac{1}{2\gamma}\|x - y\|_U^2$ is also block separable. $z_B = \text{Prox}_\gamma(x_B)$ follows from this by setting all entries of x except those in block b to 0. Finally, (6.12) follows by taking derivatives on both sides of the definition of proximal operator. ■

Let $\tilde{g}_t$ and $\tilde{u}_t$ denote the aggregated gradients and scaling coefficients at server nodes, respectively. Assume that each worker randomly skips a coordinate with probability $1 - q$, where $0 < q < 1$. Let $g_t := q^{-1}\tilde{g}_t$ and $u_t := q^{-1}\tilde{u}_t$ be the unbiased inexact gradient and scaling coefficient estimates respectively (note that more sophisticated subsampling techniques such as reservoir sampling could be employed, too).

The next step is to bound the changes of the objective function between subsequent iterations t and $t+1$ using the updates $\Delta_t = w_{t+1} - w_t$ together with the difference between g_t and $\partial \ell(w_t)$.

Lemma 5 *Let g_t be the unbiased inexact gradient aggregated by servers at time t. Under the assumptions of Theorem 2 we have*

$$\mathbf{E}\left[\ell(w_{t+1}) - \ell(w_t)\right] \leq \left(L_{var} - \frac{M_t}{\gamma_t}\right)\|\Delta_t\|^2 + \|\Delta_t\|\,\|\partial_{B_t}\ell(w_t) - \mathbf{E}\left[g_t\right]\| \tag{6.13}$$

where the expectation is taken with respect to the random skip filter.

Proof. For notation simplicity, we drop the index t for the block indicator B_t, scaling matrix U_t, learning rate γ_t and constant M_t (recall that $M_t = \min_i (U_t)_{ii}$ is the smallest coefficient-specific learning rate as induced by the Mahalanobis metric in the proximal operator).

First note that $(()\, g_t)_B = g_t$ because the gradients are computed in block B. Hence it follows that also the update Δ_t is restricted to block B. By Lemma 4 we have that

$$(\Delta_t)_B = \text{Prox}_\gamma^U\left[(w_t)_B - \gamma U^{-1} g_t\right] - (w_t)_B = \Delta_t$$

and therefore $(w_{t+1})_B = \text{Prox}_\gamma^U((w_t)_B - \gamma U^{-1} g_t)$. Using Lemma 4 again, we have

$$\frac{U}{\gamma}\left((w_t)_B - \gamma g_t - (w_{t+1})_B\right) \in \partial h((w_{t+1})_B)$$

Since h is block separable we can decompose the updates to obtain

$$\begin{aligned} h(w_{t+1}) - h(w_t) &= h((w_{t+1})_B) - h((w_t)_B) \\ &\leq \left\langle \frac{U}{\gamma}\left((w_t)_B - \gamma U^{-1} g_t - (w_{t+1})_B\right), (w_{t+1})_B - (w_t)_B \right\rangle \\ &= -\frac{1}{\gamma}\left\|\Delta_t\right\|_U^2 - \langle g_t, \Delta_t\rangle \\ &\leq -\frac{M}{\gamma}\left\|\Delta_t\right\|^2 - \langle g_t, \Delta_t\rangle \end{aligned} \tag{6.14}$$

On the other hand, only the entries of w_{t+1} in block B has been changed compared to that in w_t, which satisfies the requirement of Assumption 1, therefore, by Lemma 3,

$$\begin{aligned} \ell(w_{t+1}) - \ell(w_t) &\leq \left\langle w_{t+1} - w_t, \sum_{i=1}^m \partial_B \ell_{I_k}(w_t) \right\rangle + \sum_{i=1}^m L_{\text{var},i}\left\|\Delta_t\right\|^2 \\ &= \langle \Delta_t, \partial_B \ell(w_t)\rangle + L_{\text{var}}\left\|\Delta_t\right\|^2 \end{aligned} \tag{6.15}$$

Combining (6.14) and (6.15), we have

$$\begin{aligned} \mathbf{E}\left[\ell(w_{t+1}) - \ell(w_t)\right] &\leq \left(L_{\text{var}} - \frac{M}{\gamma}\right)\left\|\Delta_t\right\|^2 + \mathbf{E}\left[\langle \Delta_t, \partial_B \ell(w_t) - g_t\rangle\right] \\ &\leq \left(L_{\text{var}} - \frac{M}{\gamma}\right)\left\|\Delta_t\right\|^2 + \left\|\Delta_t\right\|\left\|\partial_B \ell(w_t) - \mathbf{E}\left[g_t\right]\right\| \end{aligned}$$

In other words, the amount of change between objective functions is bounded from above both by the amount of change in parameters Δ_t and by the discrepancy in the block gradient. ■

Proof of Theorem 2. We now have all ingredients to prove convergence to a stationary point. In a nutshell we must bound $\|\Delta_t\|$ and all else follows. For a fixed iteration t, let block $B = B_t$. We first upper bound the term $\|\partial_B \ell(w_t) - \mathbf{E}[g_t]\|$ in (6.13). By Assumption 1 we have for $1 \leq i \leq \tau$ that

$$\left\|\partial_B \ell_{I_k}(w_{t-i+1}) - \partial_B \ell_{I_k}(w_{t-i})\right\| \leq L_{\text{cov},k}\left\|w_{t-i+1} - w_{t-i}\right\| = L_{\text{cov},k}\left\|\Delta_{t-i}\right\|.$$

Due to the bounded delay, worker k's model is only out of date at time t in the range $t - \tau \leq t_k \leq t$. The *significantly modified* filter places an additional noise term σ_{t_k} on the model. By design of the filter we use

$$\left\|\sigma_{t_k}\right\|_\infty \leq \delta_{t_k} = \mathcal{O}\left(\frac{1}{t_k}\right).$$

Futhermore by the random skip filter, the expectation of the unbiased inexact gradient aggregated at time t is given by

$$\mathbf{E}\left[g_t\right] = \sum_{k=1}^m \partial_B \ell_{I_k}(w_{t_k} + \sigma_{t_k}).$$

Then we have

$$
\begin{aligned}
&\|\partial_B \ell(w_t) - \mathbf{E}\left[g_t\right]\| \\
=&\left\|\sum_{k=1}^{m}\sum_{i=1}^{t-t_k}\left(\partial_B \ell_{I_k}(w_{t-i+1}) - \partial_B \ell_{I_k}(w_{t-i})\right) + \partial_B \ell_{I_k}(w_{t_k}) - \partial_B \ell_{I_k}(w_{t_i}+\sigma_{t_i})\right\| \\
\leq&\sum_{k=1}^{m}\sum_{i=1}^{t-t_k}\|\partial_B \ell_{I_k}(w_{t-i+1}) - \partial_B \ell_{I_k}(w_{t-i})\| + \|\partial_B \ell_{I_k}(w_{t_k}) - \partial_B \ell_{I_k}(w_{t_k}+\sigma_{t_k})\| \\
\leq&\sum_{k=1}^{m}\sum_{i=1}^{t-t_k} L_{\text{cov},k}\left\|\Delta^{t-i}\right\| + L_{\text{cov},k}\left\|\sigma_{t_k}\right\| \\
\leq&\sum_{k=1}^{m}\sum_{i=1}^{\tau} L_{\text{cov},k}\left\|\Delta^{t-i}\right\| + L_{\text{cov},k}\sqrt{p}\delta_{t-\tau} \\
=&\sum_{i=1}^{\tau} L_{\text{cov}}\left\|\Delta^{t-i}\right\| + L_{\text{cov}}\sqrt{p}\delta_{t-\tau}
\end{aligned} \tag{6.16}
$$

where we used the fact that $\sigma_{t_k} = (\sigma_{t_k})_{B_{t_k}}$ so that Assumption 1 can be applied and $\|x\| \leq \sqrt{p}\|x\|_\infty$. Substitute (6.16) into (6.13) in Lemma 5, we have

$$
\begin{aligned}
\mathbf{E}\left[\ell(w_{t+1}) - \ell(w_t)\right] &\leq \left(L_{\text{var}} - \frac{M_t}{\gamma_t}\right)\|\Delta_t\|^2 + \sum_{i=1}^{\tau} L_{\text{cov}}\|\Delta_t\|\left(\left\|\Delta^{t-i}\right\| + \sqrt{p}\delta_{t-\tau}\right) \\
&\leq \left(L_{\text{var}} + \frac{L_{\text{cov}}\tau}{2} - \frac{M_t}{\gamma_t}\right)\|\Delta_t\|^2 + \sum_{i=1}^{\tau}\frac{L_{\text{cov}}}{2}\left\|\Delta^{t-i}\right\|^2 + L_{\text{cov}}p\delta_{t-\tau}^2
\end{aligned}
$$

Summing over t yields

$$
\mathbf{E}\left[\ell(w_{t+1}) - \ell(w_1)\right] \leq \sum_{t=1}^{T}\left(L_{\text{var}} + L_{\text{cov}}\tau - \frac{M_t}{\gamma_t}\right)\|\Delta_t\|^2 + L_{\text{cov}}p\delta_{t-\tau}^2 \tag{6.17}
$$

Define $c_t = \frac{M_t}{\gamma_t} - L_{\text{var}} - L_{\text{cov}}\tau$, and assume $\gamma_t = \frac{M_t}{L_{\text{var}}+L_{\text{cov}}\tau+\epsilon}$ for all t with $\epsilon > 0$, then all $c_t = \epsilon > 0$. So

$$
\epsilon\sum_{t=1}^{T}\|\Delta_t\|^2 \leq \sum_{t=0}^{T} c_t\|\Delta_t\|^2 \leq \mathbf{E}\left[\ell(w_1) - \ell(w_{t+1})\right] + L_{\text{cov}}p\delta_{t-\tau}^2 \tag{6.18}
$$

for any T. Since $\delta_t = O(\frac{1}{t})$, and by the fact that $1 + \frac{1}{2^2} + \frac{1}{3^2} + \ldots = \frac{\pi^2}{6}$. Then the RHS of (6.18) is constant when $T \to \infty$, which implies $\lim_{t\to\infty}\Delta_t \to 0$. So $\lim_{t\to\infty}\text{Prox}_{\gamma_t}^{U_t}(w_t) - w_t \to 0$, thus we find a local minimal point. ■

Chapter 7

EMSO: Efficient Minibatch Training for Stochastic Optimization

7.1 Introduction

7.1.1 Problem formulation

We first formally introduce the optimization problem studied in this chapter. Let $f_i : \Omega \to \mathbb{R}$ be a *convex* loss function evaluated with the i-th data example. Let w be a shared parameter which we optimize over to minimize the average loss over n examples:

$$w_* = \operatorname*{argmin}_{w \in \Omega} f(w), \qquad \text{where } f(w) = \frac{1}{n} \sum_{i=1}^{n} f_i(w). \tag{7.1}$$

7.1.2 Minibatch Stochastic Gradient Descent

Standard stochastic gradient descent (SGD) processes one example at a time, while minibatch training makes SGD easy to be parallelized for a distributed computing environment by considering a subset of examples, i.e. a minibatch, at a time. In the vanilla minibatch SGD algorithm, we choose a minibatch I with b random examples, we can define the loss function on this minibatch as

$$f_I(w) = \frac{1}{|I|} \sum_{i \in I} f_i(w). \tag{7.2}$$

Let I be uniformly distributed over all minibatches, the expected loss function is given by

$$f(w) = \mathbf{E}_I \left[f_I(w) \right].$$

Recall that Ω is the support of the variable w. In the simple case where $\Omega = \mathbb{R}^d$, the minibatch SGD employs the following stochastic updating rule. At each iteration t, we randomly pick I_t and update the variable as:

$$w_t = w_{t-1} - \eta_t \nabla f_{I_t}(w). \tag{7.3}$$

If the support Ω is a nontrivial set, a projection step is followed[156], which finds the nearest neighbor of w_t in the support.

If the loss function f_i is convex, minibatch SGD converges to the optimum at a rate $\mathcal{O}(1/\sqrt{bT}+1/T)$ [44], where T is the number of iterations. Note that given the same number of processed examples, standard SGD converges at a rate of $\mathcal{O}(1/\sqrt{bT}+1/(bT))$. If the batch size b is larger than the number of iterations, then the term $1/T$ will dominates the rate and therefore gives worse convergence speed than standard SGD. The convergence rate slowdown is indeed observed in practice especially with a large minibatch size.

There are other challenges for efficient implementation of minibatch training. In a distributed implementation, machines need to communicate with each other over iterations in order to synchronize the shared variables. Given that both the bandwidth and the latency of network communication are much worse than accessing physical memory, such synchronization cost of minibatch training may make it prohibitive for large scale applications. A larger minibatch size may serve to reduce the communication cost, however it also slows down the convergence rate in practice [26]. For the same problem, if the standard SGD converges with T iterations, the vanilla minibatch training with batch size b needs more than T/b iterations. With a larger batch size, such increase of computation workload may outweigh the benefits of the reduced communication cost. In addition, the I/O costs also increases if the data is too large to fit into memory so that one need to fetch the minibatch from disk or network for more iterations [150].

7.1.3 Related Work and Discussion

The idea of using minibatch in stochastic optimization has been studied by a number of researchers. For example, it was shown in [44] that distributed minibatch gradient can achieve a convergence rate of $\mathcal{O}(1/\sqrt{Tb}+1/T)$, which is comparable to that of serial SGD when the minibatch size is small. Additional studies include [38, 120, 135]. There is also a large volume of works aiming to improve the standard minibatch approach. For example, [77] proposed to solve $\min_w f_{I_t}(w)$ directly, while [25] presented a L-BFGS style updating. In addition, [72, 93] argued to reduce the stochastic variance via gradients computed on the whole dataset.

There exist a lot of attempts to speed up the convergence of minibatch SGD. For example, in [58], asynchronous communication is suggested. In [120], the accelerated version of minibatch SGD is studied. At a more fundamental level, the authors of [94, 157] consider the problem of solving subproblems in parallel, followed by averaging of the results. The above works, however, are not the most efficient in the sense that no communication occurs during the phase of gradients computation.

Another line of research focuses on the practical performance, especially when data cannot fit into memory. For example, in [150] the approach of solving linear SVM in the dual form by processing a block of data at each time was studied. In [96] it showed that having both I/O and computational threads working together can further improve the performance. In [30] the selection of the data blocks was explored.

7.1.4 Our work

We propose an efficient minibatch algorithm with reduced communication cost and good convergence properties. In particular, the new algorithm does not slow down the convergence as the minibatch size increases. The key observation is that, when a minibatch is large, it is desirable to solve a more complex optimization problem, rather than simply update the solution using the stochastic gradients. More specifically, in each iteration, we *solve* a conservative risk minimization subproblem, which consists of two components: the original objective function on the minibatch and a conservative penalty. In this way, we are able to gain more progress on the objective function from each minibatch iteration. The conservative penalty reduces the variance and prevents divergence from the previous consensus. The key challenge of this approach is twofold: we need a more sophisticated update strategy as well as an efficient way to solve the conservative subproblem, such that the increase in computation workload does not offset the reduced synchronization cost.

Our approach differs from the previous works by exploiting the data partition in a nontrivial manner beyond gradient flow methods– we solve subproblems using the partition data instead. We show that the proposed algorithm has a $\mathcal{O}(1/\sqrt{bT})$ convergence rate, which significantly improves the result in [44] when the batch size b is large. We show that it can be further improved to $\mathcal{O}(\log T/(\lambda bT) + \lambda/(\sqrt{b}T))$ for a λ-strongly convex objective function. We also show a communication efficient distributed implementation of the proposed algorithm, and demonstrate the efficacy of the proposed algorithm with experiments on large scale datasets.

7.2 Efficient Minibatch Training Algorithm

7.2.1 Our algorithm

Throughout this chapter, we focus on the case where $f_i(w)$ is convex for all i. This standard assumption simplifies the theoretical analysis. The proposed algorithm, however, can be applied to more general non-convex functions.

First note that we can write the updating rule of minibatch SGD as solving the following optimization problem:

$$w_t = \operatorname*{argmin}_{w\in\Omega}\Big[f_{I_t}(w_{t-1}) + \langle \nabla f_{I_t}(w_{t-1}), w - w_{t-1}\rangle + \frac{1}{2\eta_t}\left\|w - w_{t-1}\right\|_2^2\Big],$$

where η_t is the learning rate, I_t is the indexes of the examples choose at iteration t.

Note that this objective can also be viewed as an approximate solution to the minimization of the loss function $f_{I_t}(w)$ plus a conservative penalty in w_{t-1}, in the sense that it is using the first order Taylor approximation of $f_{I_t}(w)$ at w_{t-1}. However, this first order approximation might be too coarse to make sufficient progress in each iteration. In particular, such an aggressive tradeoff of fast convergence in favor of computational efficiency is undesirable for a large batch size, as there is often significant overhead of switching minibatches, due to process synchronization, data reads from disk, and network communication. Also, note that given the variance of the randomly chosen examples, SGD often favors a small step size, and thus each iteration makes

a even smaller progress. However, when the size of a minibatch increases, the variance of a minibatch of examples decreases, which suggests that more sophisticated methods could be used towards faster convergence rate.

The proposed algorithm is given in Algorithm 6. The key step is that at each iteration the parameter is updated by solving the following subproblem:

$$w_t = \operatorname*{argmin}_{w \in \Omega} \left[f_{I_t}(w) + \frac{\gamma_t}{2} \|w - w_{t-1}\|_2^2 \right]. \tag{7.4}$$

Note that the objective function in the optimization has two components. The first part is the loss function on minibatch I_t, which serves to exploit this minibatch as much as possible; the second component is a conservative constraint which limits dramatic changes of the parameter to avoid over-utilization of this minibatch.

Compared to the update rule of SGD, here we need to solve the more complex conservative subproblem for each minibatch. We assume that the optimization is performed exactly for the simplicity of convergence analysis; however in practice, an approximate solution to the subproblem suffices, especially in the early stages of the iterations. If the computational cost for this approximate optimization is not too high, this update rule can speed up the convergence and drastically reduce the amount of network communication required between iterations.

Algorithm 6 Single node template for EMSO

1: **Input:** Initial w_0, conservative coefficients $\gamma_1, \ldots, \gamma_T$.
2: **for** $t = 1, \ldots, T$ **do**
3: randomly choose minibatch $I_t \subset \{1, \ldots, n\}$ of size b
4: solve the conservative subproblem:

$$w_t = \operatorname*{argmin}_{w \in \Omega} \left[f_{I_t}(w) + \frac{\gamma_t}{2} \|w - w_{t-1}\|_2^2 \right].$$

5: **end for**

7.2.2 Convergence Analysis

Compared to the vanilla minibatch SGD, the advantage of our algorithm is that the convergence does not slow down when the minibatch size increases. We first introduce the notion of a Bregman divergence for convex functions f defined as:

$$D_f(w; w') := f(w) - f(w') - \nabla f(w')^\top (w - w'). \tag{7.5}$$

This is the difference between $f(w)$ and the value of the first-order Taylor expansion of f at w', when evaluated at w. The properties of Bregman divergence include:

Non-negativity: $D_f(w; w') \geq 0$.

Convexity: $D_f(w; w')$ is convex with respect to w.

Linearity: $D_f(w; w')$ is linear with respect to f, namely

$$D_{f+cf'}(w; w') = D_f(w; w') + cD_{f'}(w; w').$$

We also impose the following technical assumption, which bounds the amount of "surprise" we can expect on the full Bregman divergence when it is replaced by the Bregman divergence on each minibatch plus a conservative penalty.

Assumption 6 *Assume that for all t:*

$$\mathbf{E}_{I_t}\left[D_f(w_t; w_{t-1})\right] \leq \mathbf{E}_{I_t}\left[D_{f_{I_t}}(w_t; w_{t-1}) + \frac{\gamma_t}{2}\left\|w_t - w_{t-1}\right\|_2^2\right].$$

Note that in general, $D_f(w; w_{t-1}) = \mathbf{E}_{I_t}\left[D_{f_{I_t}}(w; w_{t-1})\right]$ holds for w that is independent of the minibatch I_t. However, since the parameter w_t is a function of I_t, we need to impose this assumption for some $\gamma_t > 0$. This assumption holds as long as we pick γ_t greater than or equal to the smoothness parameter of f. Namely,

$$f(w) - f(w') - \nabla f(w')^\top (w - w') \leq \frac{\gamma_t}{2}\|w - w'\|_2^2.$$

In other words, the counterpart of strong convexity, namely that there exists a quadratic *upper* bound on the amount of change, suffices to guarantee this assumption. In practice, however, one can be more aggressive to allow a much smaller γ_t when the minibatch size is large. In fact, one can show that a choice of $\gamma_t = O(1/b)$ is sufficient.

We state the main theorem for convergence analysis below. The detailed proof is deferred to Section 7.4.

Theorem 7 *Consider the stochastic update rule* (7.4)*. Assume that f_i is λ-strongly convex for all i. Under Assumption 6 with the step-size $\gamma_t = \gamma + \lambda(t-1)$, for all parameter $w^* \in \Omega$, it holds that:*

$$\sum_{t=1}^{T} \mathbf{E}[f(w_t) - f(w^*)] \leq \frac{\gamma}{2}\|w^* - w_0\|_2^2 + \frac{A^2}{b}\sum_{t=1}^{T}\frac{1}{\gamma_t},$$

where

$$A^2 = \sup_{w \in \Omega} n^{-1} \sum_{i=1}^{n} \|\nabla f_i(w) - \nabla f(w)\|_2^2.$$

For general convex functions, we set $\lambda = 0$, and this amounts to a constant update rate γ in the above theorem. In this case, setting $\gamma = \sqrt{\frac{2T}{b}}\frac{A}{\|w^* - w_0\|_2}$ minimizes the right hand side of the bound. Note that there is no a-priori guarantee that such a small γ is feasible. However, since the variance decreases with minibatch size in the order of $\mathcal{O}(1/b)$, the scaling of $\gamma = \mathcal{O}(1/\sqrt{b})$ is appropriate. When it is feasible, this yields the following aggregate regret bound:

$$\frac{1}{T}\sum_{t=1}^{T} \mathbf{E}[f(w_t) - f(w^*)] \leq \frac{\sqrt{2}A}{\sqrt{Tb}}\|w^* - w_0\|_2.$$

The above bound says that if minibatch size is b, after T steps, we have a convergence rate in the order of $\mathcal{O}(1/\sqrt{Tb})$. Therefore increasing minibatch size does not affect convergence rate in terms of the number of training examples processed by the algorithm. Compared to the standard minibatch SGD, the convergence rate $\mathcal{O}(1/\sqrt{bT}+1/T)$ is dominated by the second term $O(1/T)$ when the number of iterations T is less than the batch size b, which often happens for using large batch size with early stop.

In addition, for strongly convex loss function with $\lambda > 0$, we can optimize γ and achieve a stronger regret bound $\mathcal{O}(\log T/(\lambda bT) + \lambda/(\sqrt{b}T))$.

7.2.3 Efficient Implementation

In this section, we discuss two distributed implementations of the proposed algorithm. Note that we solve the conservative subproblem (7.4) exactly only for the simplicity of the convergence analysis. In practice, we only need to solve this optimization approximately.

Early Stopping Many optimization methods for the original problem (7.1) can be applied to the subproblem in (7.4), and the most suitable method indeed depends on the specific risk function. Here we discuss two generic methods that are helpful for distributed implementation of (7.4), and we use early stopping to speed up the computation.

Algorithm 7 EMSO-GD: solving (7.4) with gradient descent

1: **Input:** model parameter w_{t-1}, minibatch I_t
2: conservative coefficient γ_t, learning rate η_t
3: **Output:** updated model parameter w_t
4: $w \leftarrow w_{t-1}$
5: **for** $\ell = 1, \ldots, L$ **do**
6: update

$$w \leftarrow w - \eta_t \left(\nabla f_{I_t}(w) + \gamma_t(w - w_{t-1})\right) \tag{7.6}$$

7: **end for**
8: $w_t \leftarrow w$

Our first method, named EMSO-GD, is a direct extension of SGD. Note that, if we set the parameter $\gamma_t = 0$ in (7.4), SGD would be equivalent to performing gradient descent with a single pass of the minibatch with w_{t-1} as the start point. We can relax the single pass constraint to obtain a more accurate solution of the conservative subproblem. In Algorithm 7 we sketch the algorithm EMSO-GD. It solves the subproblem (7.4) using gradient descent. We limit the maximal number of iterations to a fixed constant L. This early stopping strategy helps to simplify the synchronization in the distributed implementation.

Our second method EMSO-CD, sketched in Algorithm 8, is motivated by the work in [150], which applies minibatch coordinate descent to solve the dual form of linear SVM. For our subproblem, we directly solve it using coordinate descent in the *primal* form. Let p denote the dimension of parameter w. In each iteration, EMSO-CD chooses a random coordinate $j \in [1, p]$,

Algorithm 8 EMSO-CD: solving (7.4) with coordinate descent

Input: previous parameter w_{t-1}, minibatch I_t
conservative coefficient γ_t
Output: new parameter w_t
$w \leftarrow w_{t-1}$
for $\ell = 1, \ldots, Lp$ **do**
    randomly choose a coordinate $j \in [0, p]$
    update

$$[w]_j \leftarrow [w]_j - \eta_t \frac{[\nabla f_{I_t}(w)]_j + \gamma_t([w]_j - [w_{t-1}]_j)}{[\nabla^2 f_{I_t}(w_t)]_{jj} + \gamma_t} \tag{7.7}$$

end for
$w_t \leftarrow w$

and solves the following one dimensional problem using Newton's method:

$$\underset{[w]_j}{\operatorname{argmin}} \left\{ f_{I_k}(w) + \frac{\gamma_t}{2} \|w - w_{t-1}\|_2 \right\},$$

We apply early stopping similar to that in EMSO-GD.

Distributed Model Averaging To implement our algorithm in a distributed computing environment, we first partition the minibatch into m parts, and then assign them to different machines. Instead of having all machines coordinate with each other to solve the subproblem, which is expensive given the limited communication resources, we propose a more communication friendly approach. In Algorithm 9 each machine simply solves the subproblem independently using the part of data assigned to it, and then the results are averaged across all machines at the end of each iteration.

Algorithm 9 Distributed EMSO with model averaging.

Input: initialization w_0, conservative coefficients $\{\gamma_t\}_{t=1}^T$, learning rate $\{\eta_t\}_{t=1}^T$, number of machines k
for $t = 1, \ldots, T$ **do**
    randomly choose minibatch I_t
    partition $I_t = \bigcup_{k=1}^m I_t^{(k)}$
    for $k = 1, \ldots, m$ **do** in parallel
        solve the conservative subproblem on $I_t^{(k)}$ using Algorithm 7 or 8 to obtain $w_t^{(k)}$
    end for
    average $w_t = \frac{1}{m} \sum_{k=1}^m w_t^{(k)}$ via communication
end for

name	update	iteration	distributed
L-BFGS	[89]	batch	yes
LIBLINEAR	[51]	batch	no
Minibatch SGD	(7.3)	minibatch	yes
EMSO-GD	(7.6)	minibatch	yes
EMSO-CD	(7.7)	minibatch	yes

Table 7.1: Evaluated Algorithms.

7.3 Experiments

In this section, we evaluate our algorithm and its implementation with a case study of classification using logistic regression. The loss function is given by $f_i(w) = \log(1 + \exp(-y_i\langle x_i, w\rangle))$. We compare our algorithms with two other algorithms. The results are summarized in Table 7.1. All experiments were carried on the cluster CampusB, where each machine is equipped with four AMD Opteron Interlagos 16 core 6272 CPUs, 128GB memory and 10Gbit Ethernet. We summarize the key implementation details below.

L-BFGS This is a parallelized version of the classical memory-limited BFGS method, as described in [89]. The root machine obtains subgradients from each of the client machines and aggregate the information into a global subgradient. Then the parameters are updated and is broadcasted to the machines.

LibLinear It is a sophisticated batch solver for convex problems. This single-machine implementation is obtained from the author's website[1]. It is included as a reference to existing well-known solvers.

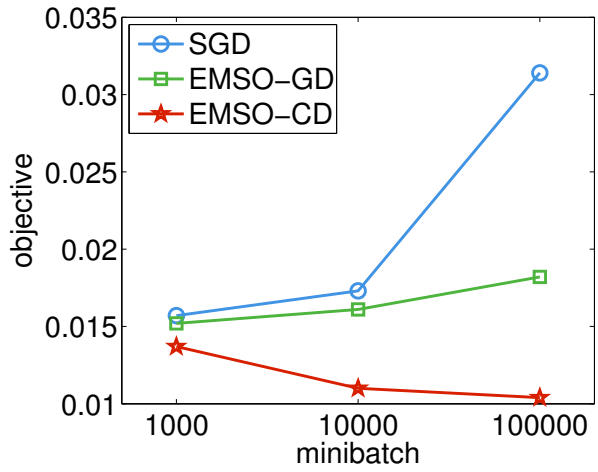
Minibatch SGD As the name of this algorithm suggests, for each minibatch, it computes the subgradients using the machines, and then aggregate the subgradients to a full minibatch subgradient. After that, we update the parameter on the root machine using (7.3) and broadcast the changes. We used an $\mathcal{O}(1/\sqrt{t})$ decay learning rate to set $\eta_t = \eta\sqrt{\alpha/(t+\alpha)}$ for the t-th iteration, where the constants η and α specify the initial scale and decaying speed, respectively. We use grid search in the range of $\eta \in \{10^0, \ldots, 10^{-5}\}$ and $\alpha \in \{10^0, \ldots, 10^4\}$ to choose the best constants to optimize the convergence progress.

EMSO-GD We use the parameter-averaging approach introduced in Algorithm 9. Recall that this algorithm uses the higher order information of the loss function when solving a subproblem with each minibatch.

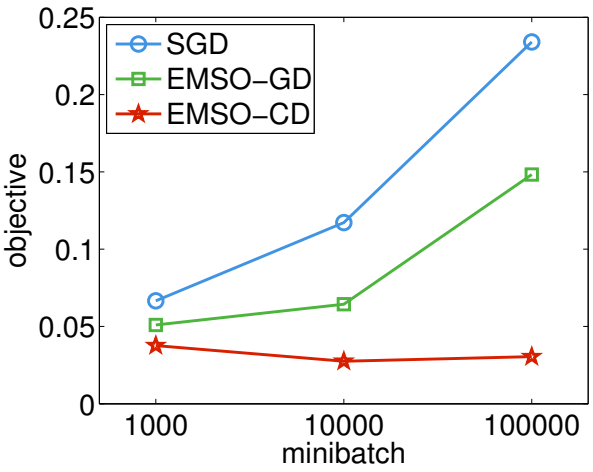
EMSO-CD The only difference between this algorithm and EMSO-GD is that it uses coordinate descent to update the parameters. For these two EMSO variants, we set $\lambda = 0$, an optimize γ in the range $\{10^0, \ldots, 10^5\}$.

Minibatch Size and Convergence A first sanity check is to ascertain the convergence results of minibatch methods. For this purpose we set the batch size to 10^3, 10^4, and 10^5 and examine the value of the objective function after processing 10^7 examples. Figure 7.2 shows that the value

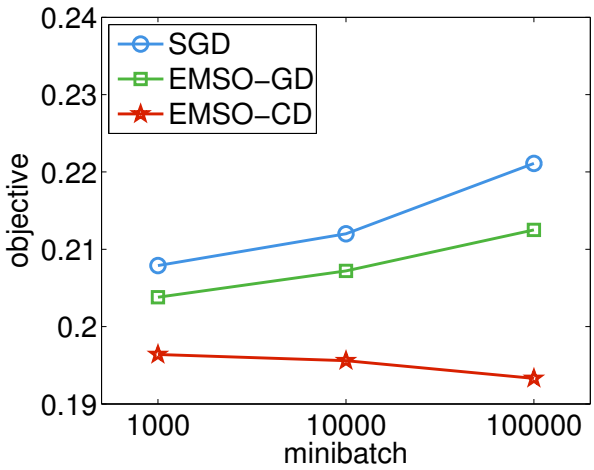
[1] `http://www.csie.ntu.edu.tw/~cjlin/liblinear/`



(a) Dataset: KDD14



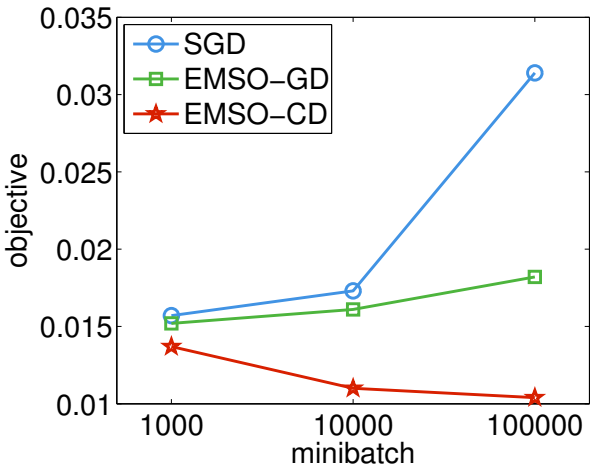
(b) Dataset: URL



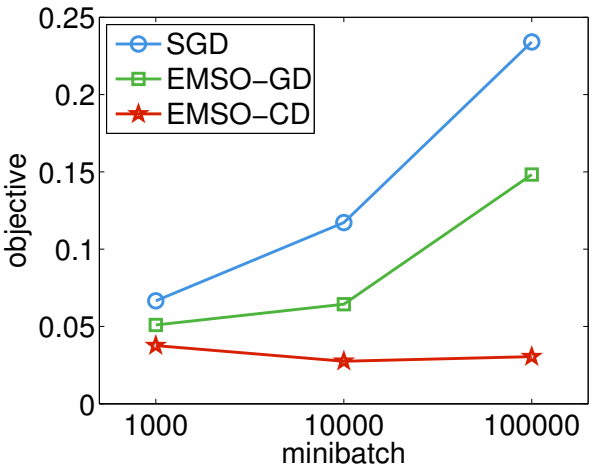
(c) Dataset: CTRa

Figure 7.1: Objective value versus minibatch size after in total 10^7 examples are processed in a single node. Here CTRa is downsampled to 4 millions examples due to the limited capacity of a single node.

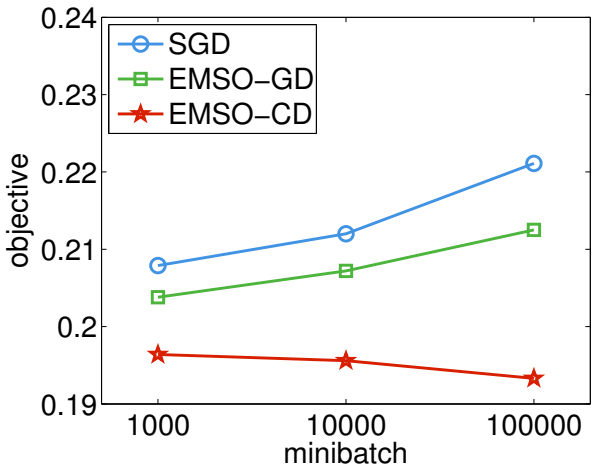
of the objective function for minibatch SGD significantly increases with the minibatch size as it does not efficiently use the minibatches. This problem is the worst on the dataset KDD14, which is dense and extremely unbalanced in terms of the labels. Our algorithm EMSO-GD, which performs 5 iterations of gradient descent in a minibatch, shows a much better convergence when the batch size increases as it makes a better use of the minibatch by potentially extracting more information from the multiple iterations. Moreover, Figure 7.2 shows that the convergence is even more stable when solving the conservative subproblems using coordinate descent. A possible explanation is that, even though each minibatch is processed only twice, in EMSO-CD, the parameter from previous iteration serves as a warm start to improve the solution quality of the conservative subproblem. It seems that this advantage of coordinate descent can offset the



(a) Dataset: KDD14



(b) Dataset: URL

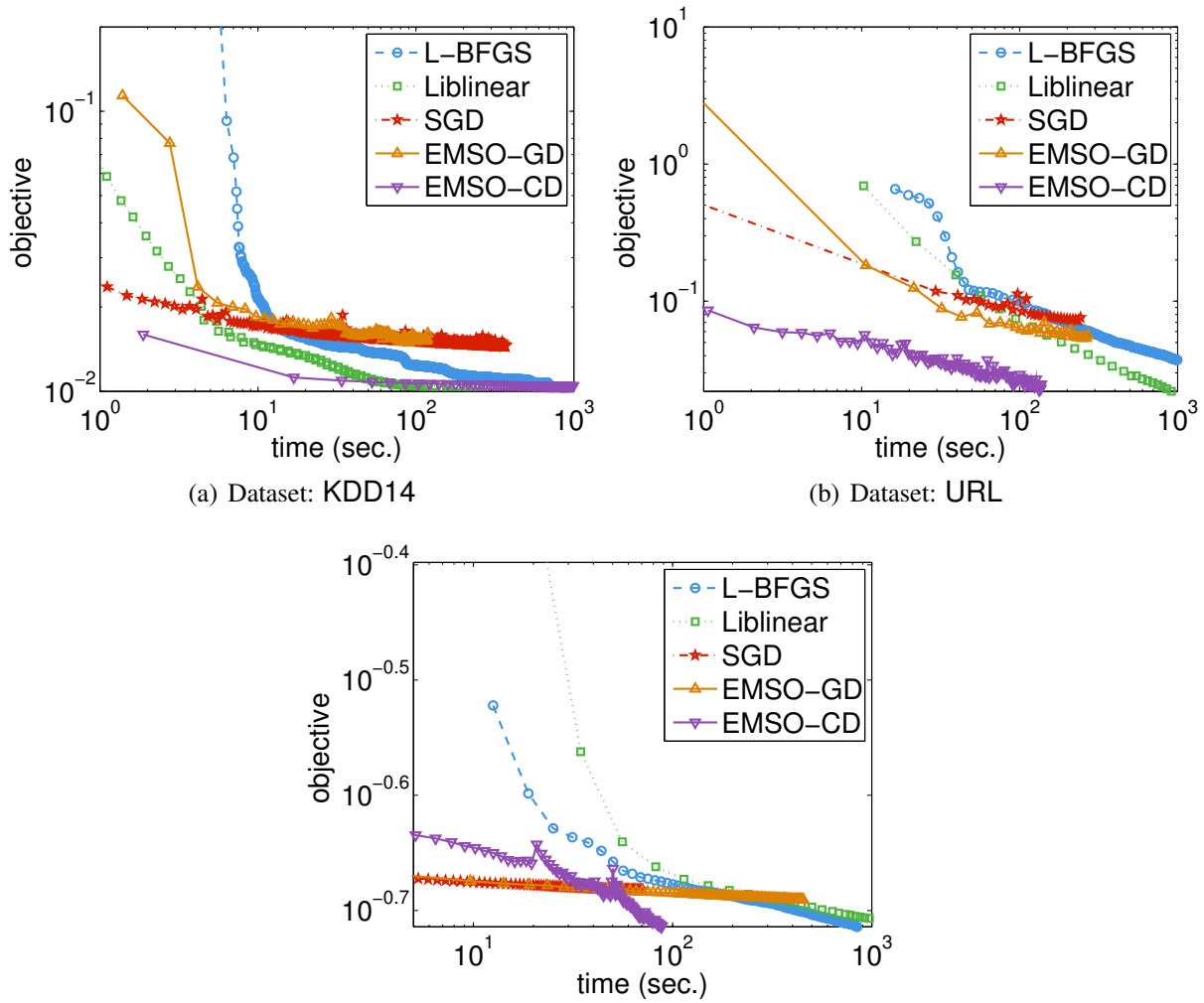


(c) Dataset: CTRa (downsampled to 4 millions examples due to the limited capacity of a single machine)

Figure 7.2: Value of the objective function versus minibatch size after in total 10^7 examples are processed on each machine.

negative effect of minibatch on convergence.

Run time and Convergence We compare the five algorithms listed in Table 7.1 by tracking the value of the objective function over time. We use a batch size of 10^5 for EMSO-CD and 10^3 for the rest. Figure 7.3 shows that the convergence behavior of L-BFGS and LibLinear are similar: slow at the beginning and faster towards the end. Observe that EMSO-GD is comparable to SGD. Note that even though EMSO-GD converges faster in terms of the number of minibatch iterations, it requires 5 times more computational time than SGD. Notably, even with a larger minibatch size, EMSO-CD is at least 10 times faster than the other algorithms.



(a) Dataset: KDD14

(b) Dataset: URL

(c) Dataset: CTRa

Figure 7.3: Value of the objective function versus run time.

Minibatch Size and the Synchronization Cost Recall that one advantage of using a large minibatch size is the potential reduction in synchronization cost of communication between machines. Figure 7.4 shows the proportion of synchronization cost in the overall runtime. The proportion is considerable even with just 12 machines, and decreases with the minibatch size. This is due to fact that both EMSO-GD and EMSO-CD solve a more complex optimization problem with each minibatch, and thus the amount of actual computation between the synchronization passes increases. In addition, although EMSO-CD passes a minibatch just twice in our experiment, it requires significant more exponentiation operations than EMSO-GD (a fast special functions library would probably address this issue). As a result, it consumes more CPU time in actual computation than EMSO-GD, and has a smaller fraction of synchronization cost.

In Figure 7.5 we compare the convergence results between EMSO-CD and L-BFGS with various minibatch sizes. In the left subfigure, we fix the total number of examples processed to be 5×10^6. Similar to the single machine results in Figure 7.2, EMSO-GD slightly improves SGD

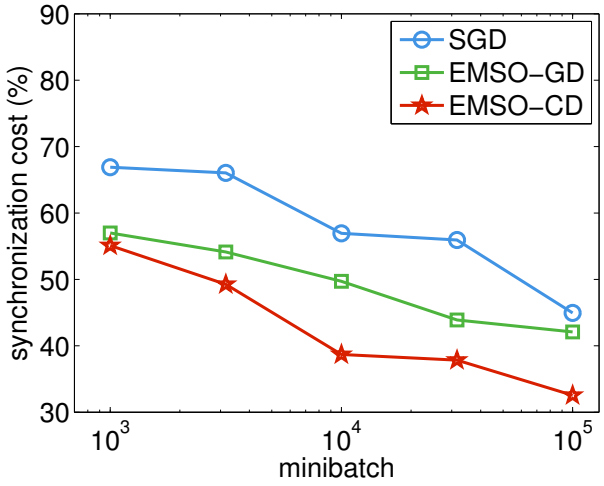


Figure 7.4: The fraction of synchronization cost as a function of minibatch size when using 12 machines.

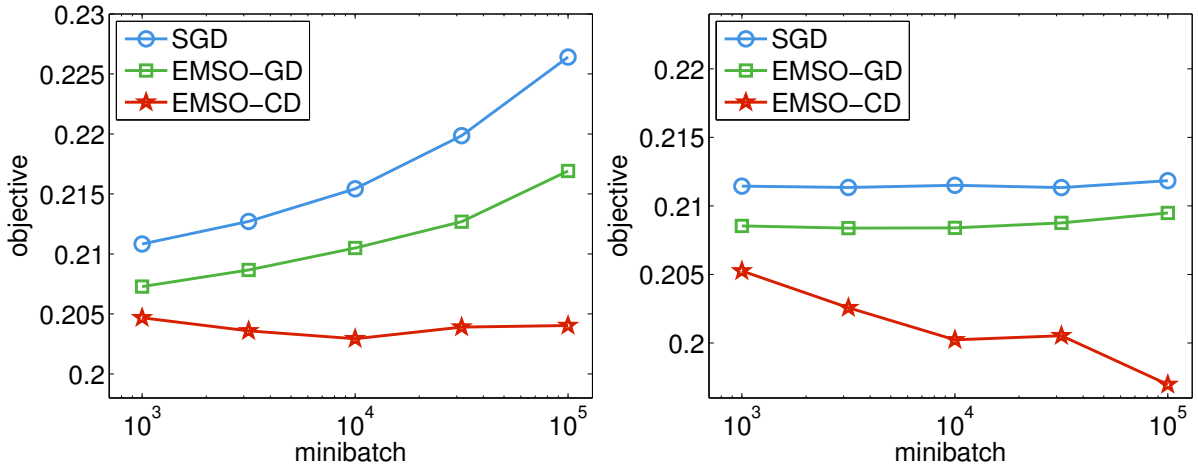


Figure 7.5: Value of the objective function versus minibatch size. 12 machines are used. Left: the total number of examples is fixed to 5×10^6. Right: the runtime is fixed to 1000 seconds.

while EMSO-CD is much better than the others. In the right subfigure, we fix the run time to be $1,000$ seconds. In this setting, with a large minibatch size, the proportion of synchronization cost decreases and more time is allocated to the actual computation. In particular, there is a clear advantage for EMSO-CD to use a large batch size in distributed computing.

Scalability We conclude our experimental evaluation by assessing the performance for varying numbers of machines in the distributed computing environment, with a comparison between EMSO-CD and L-BFGS shown in Figure 7.6. We can see that both algorithms benefit from the increase of the number of machines, while L-BFGS gains more than EMSO-CD. This is because L-BFGS passes the whole training data in each iteration and thus the portion of synchronization cost is only 15% compared to 30% of EMSO-CD. However, EMSO-CD is still 10 times faster

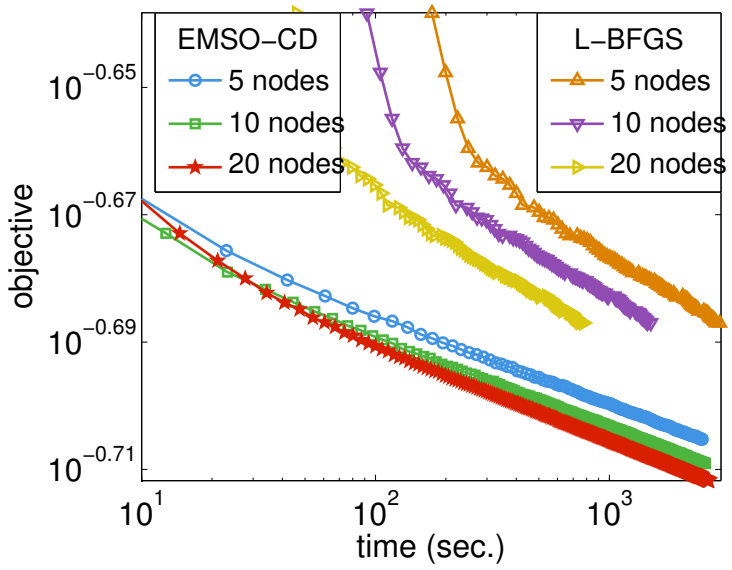


Figure 7.6: Value of the objective function versus run time for EMSO-CD and L-BFGS using different numbers of machines.

	objective = 0.2		objective = .1972	
#nodes	time	speedup	time	speedup
5	879s	1.00x	2439s	1.00x
10	499s	1.76x	1367s	1.78x
20	363s	2.42x	962s	2.54x

Table 7.2: Run time and speedup for EMSO-CD to reach the same value of the objective function when running on 5, 10 and 20 machines.

than L-BFGS for all possible number of machines. Table 7.2 shows the speedup for EMSO-CD to reach specific objective values. When the number of nodes doubles from 5 to 10, there is an 1.75x speedup on average, and when the nodes number increases by 4 times, we have a 2.5x speedup.

7.4 Proof of Theorem 7

For convenience, we define the regularized minibatch loss by $h_t(w) = f_{I_t}(w) + \gamma_t \|w\|_2^2/2$. Our proof relies on three lemmas. First, we upper bound $\|w_t - \bar{w}_t\|_2$, where $\bar{w}_t$ is similar to w_t except for optimizing over all examples. That is, the gradients differ via $\|\nabla f(\bar{w}_t) - \nabla f_{I_t}(\bar{w}_t)\|_2$. Next, we show that the expectation of the latter, namely the variance of gradient over a minibatch, is bounded from above by A^2/b. Finally, we characterize the progress from time $t-1$ to t.

Lemma 8 *Let $\bar{w}_t$ be the minimizer of the conservative version of the expected risk, namely*

$$\bar{w}_t = \operatorname*{argmin}_{w \in \Omega} \left[f(w) + \frac{\gamma_t}{2} \|w - w_{t-1}\|_2^2 \right]. \tag{7.8}$$

We can bound the difference between the solution $\bar{w}_t$ and the solution w_t obtained with a mini-batch as in (7.4) *by:*

$$\|w_t - \bar{w}_t\|_2 \leq \frac{1}{\gamma_t} \|\nabla f(\bar{w}_t) - \nabla f_{I_t}(\bar{w}_t)\|_2 .$$

Proof. Since $w_t = \operatorname{argmin}_{w \in \Omega} h_t(w)$, we have from the first order KKT condition at w_t:

$$\nabla h_t(w_t)^\top (w_t - \bar{w}_t) \leq 0$$

In addition, the first order KKT condition of (7.8) at $\bar{w}_t$ combined with the fact that $h_t(w) = f_{I_t}(w) + \frac{\gamma_t}{2} \|w - w_{t-1}\|_2^2$ implies that

$$(\nabla h_t(\bar{w}_t) + \nabla f(\bar{w}_t) - \nabla f_{I_t}(\bar{w}_t))^\top (w_t - \bar{w}_t) \geq 0.$$

By substracting the first inequality from the second inequality, and rearranging terms, we obtain:

$$\begin{aligned} &(\nabla h_t (w_t) - \nabla h_t (\bar{w}_t))^\top (w_t - \bar{w}_t) \\ \leq &(\nabla f (\bar{w}_t) - \nabla f_{I_t} (\bar{w}_t))^\top (w_t - \bar{w}_t) . \end{aligned} \tag{7.9}$$

By additivity of Bregman divergences we have

$$\begin{aligned} D_{h_t}(\bar{w}_t; w_t) &= D_{f_{I_t}}(\bar{w}_t; w_t) + \frac{\gamma_t}{2} \|\bar{w}_t - w_t\|_2^2 . \\ \text{hence } D_{h_t} (\bar{w}_t; w_t) &\geq \frac{\gamma_t}{2} \|\bar{w}_t - w_t\|_2^2 . \end{aligned}$$

Similarly $D_{h_t}(w_t; \bar{w}_t) \geq \frac{\gamma_t}{2} \|\bar{w}_t - w_t\|_2^2$. It follows that

$$\begin{aligned} \gamma_t \|w_t - \bar{w}_t\|_2^2 &\leq D_{h_t}(\bar{w}_t; w_t) + D_{h_t}(w_t; \bar{w}_t) \\ &= (\nabla h_t(w_t) - \nabla h_t(\bar{w}_t)^\top (w_t - \bar{w}_t) \\ &\leq (\nabla f(\bar{w}_t) - \nabla f_{I_t}(\bar{w}_t))^\top (w_t - \bar{w}_t) \\ &\leq \|\nabla f(\bar{w}_t) - \nabla f_{I_t}(\bar{w}_t)\|_2 \|w_t - \bar{w}_t\|_2, \end{aligned}$$

where the second inequality is due to (7.9). The third inequality is Cauchy-Schwarz inequality. ■

Lemma 9 *Assume that we randomly choose a minibatch I of size b. We can bound the deviation between the gradient of the risk minibatch and the actual risk by:*

$$\mathbf{E}_I \left[\|\nabla f_I(w) - \nabla f(w)\|_2^2 \right] = \frac{n-b}{n-1} \frac{B^2}{b} \leq \frac{A^2}{b},$$

where $B^2 = \frac{1}{n} \sum_{i=1}^{n} \|\nabla f_i(w) - \nabla f(w)\|_2^2$.

Proof. This bound is essentially a conversion of variances from a minibatch I to the full set when using sampling without replacement. To simplify notation we use the abbreviation of $\psi_i := \nabla f_i(w) - \nabla f(w)$ and $\psi_I = \nabla f_I(w) - \nabla f(w)$. Note that by construction

$$\mathbf{E}_i \left[\psi_i\right] = \mathbf{E}_I \left[\psi_I(w)\right] = \psi \tag{7.10}$$

and therefore $B^2 = \mathbf{E}_i \left[\|\psi_i\|^2\right]$ and $B^2 \leq A^2$. The latter inequality follows since A^2 is a uniform upper bound on the variance over all $w \in \Omega$. This yields

$$\begin{aligned}
\mathbf{E}_I \left[\|\psi_I\|_2^2\right] &= \mathbf{E}_I \left[\left\|\frac{1}{b}\sum_{i\in I}\psi_i\right\|_2^2\right] = \frac{1}{b^2}\mathbf{E}_I \left[\sum_{i,j\in I}{\psi_i}^\top \psi_j\right] \\
&= \frac{1}{b^2}\mathbf{E}_I \left[\sum_{i\neq j\in I}{\psi_i}^\top \psi_j\right] + \frac{B^2}{b} \\
&= \frac{b-1}{bn(n-1)}\sum_{i\neq j}{\psi_i}^\top \psi_j + \frac{B^2}{b} \\
&= \frac{b-1}{bn(n-1)}\sum_{i,j}\psi_i^\top \psi_j + \frac{B^2}{b} - \frac{B^2}{b}\frac{b-1}{n-1} \\
&= 0 + \frac{B^2}{b}\frac{n-b}{n-1} < \frac{A^2}{b}
\end{aligned}$$

The last equality used the fact that ψ_i has zero-mean. ■

Lemma 10 *For each iteration, we can bound the expected improvement in terms of Bregman divergence by*

$$\mathbf{E}\left[D_{h_t}(w^*, w_t) - D_{h_t}(w^*, w_{t-1})\right] \leq f(w^*) - \mathbf{E}\left[f(w_t)\right] - \mathbf{E}\left[D_f(w^*; w_{t-1})\right] + \frac{1}{\gamma_t}\frac{A^2}{b}. \tag{7.11}$$

Proof. We have

$$\begin{aligned}
&D_{h_t}(w^*, w_t) - D_{h_t}(w^*, w_{t-1}) \\
=\ &D_{h_t}(w_{t-1}, w_t) + \left(\nabla h_t(w_{t-1}) - \nabla h_t(w_t)\right)^\top (w^* - w_t) \\
&+ \left(\nabla h_t(w_{t-1}) - \nabla h_t(w_t)\right)^\top (w_t - w_{t-1}) \\
\leq\ &D_{h_t}(w_{t-1}, w_t) + \nabla f_{I_t}(w_{t-1})^\top (w^* - w_t) \\
&+ \left(\nabla h_t(w_{t-1}) - \nabla h_t(w_t)\right)^\top (w_t - w_{t-1}) \\
=\ &f(w^*) - f(w_t) - D_f(w^*; w_{t-1}) \\
&- \left(\nabla f(w_{t-1}) - \nabla f_{I_t}(w_{t-1})\right)^\top (w^* - w_t) \\
&+ \left(D_f(w_t; w_{t-1}) - D_{h_t}(w_t; w_{t-1})\right),
\end{aligned}$$

where the equalities follow from algebraic manipulations and the definition of Bregman divergence; in the inequality, we used the first order KKT condition of (7.4) at w_t, implying that

$$\begin{aligned}&(\nabla f_{I_t}(w_{t-1}) + \nabla h_t(w_t) - \nabla h_t(w_{t-1}))^\top (w_* - w_t)\\ =\ &\nabla h_t(w_t)^\top (w_* - w_t) \geq 0.\end{aligned}$$

Taking expectation, we have

$$\begin{aligned}&\mathbf{E} D_{h_t}(w^*, w_t) - \mathbf{E} D_{h_t}(w^*, w_{t-1})\\ \leq\ & f(w^*) - \mathbf{E} f(w_t) - \mathbf{E} D_f(w^*; w_{t-1})\\ &- \mathbf{E}(\nabla f(w_{t-1}) - \nabla f_{I_t}(w_{t-1}))^\top (w^* - w_t)\\ &+ \mathbf{E}(D_f(w_t; w_{t-1}) - D_{h_t}(w_t; w_{t-1}))\\ \leq\ & f(w^*) - \mathbf{E} f(w_t) - \mathbf{E} D_f(w^*; w_{t-1})\\ &- \mathbf{E}(\nabla f(w_{t-1}) - \nabla f_{I_t}(w_{t-1}))^\top (w^* - w_t)\\ =\ & f(w^*) - \mathbf{E} f(w_t) - \mathbf{E} D_f(w^*; w_{t-1})\\ &- \mathbf{E}(\nabla f(w_{t-1}) - \nabla f_{I_t}(w_{t-1}))^\top (\bar{w}_t - w_t),\end{aligned} \tag{7.12}$$

where the second inequality follows from $\mathbf{E}(D_f(w_t; w_{t-1}) - D_{h_t}(w_t; w_{t-1})) \leq 0$, which is a consequence of Assumption 6. The equality holds because

$$\begin{aligned}&\mathbf{E}(\nabla f(w_{t-1}) - \nabla f_{I_t}(w_{t-1}))^\top w^*\\ =\ &\mathbf{E}(\nabla f(w_{t-1}) - \nabla \mathbf{E}_{I_t|w_{t-1}} f_{I_t}(w_{t-1}))^\top w^*\\ =\ &0 = \mathbf{E}(\nabla f(w_{t-1}) - \nabla f_{I_t}(w_{t-1}))^\top \bar{w}_t.\end{aligned}$$

Note further that

$$\begin{aligned}&- \mathbf{E}(\nabla f(w_{t-1}) - \nabla f_{I_t}(w_{t-1}))^\top (\bar{w}_t - w_t)\\ \leq\ & \sqrt{\mathbf{E}\|\nabla f(w_{t-1}) - \nabla f_{I_t}(w_{t-1}))\|_2^2\ \mathbf{E}\|\bar{w}_t - w_t\|_2^2}\\ \leq\ & \sqrt{\mathbf{E}\|\nabla f(w_{t-1}) - \nabla f_{I_t}(w_{t-1}))\|_2^2\ \mathbf{E}\|\nabla f(\bar{w}_t) - \nabla f_{I_t}(\bar{w}_t)\|_2^2}/\gamma_t\\ \leq\ & A^2/(\gamma_t b),\end{aligned}$$

where the first inequality follows from Cauchy-Schwarz inequality, the second inequality is due to Lemma 8, and the third inequality is due to Lemma 9. Plugging the above estimate into (7.12), we obtain the desired bound. ■

Finally we can prove Theorem 7 below.

Proof of Theorem 7. Under the assumption that f_i is λ-strongly convex, it follows by construction that h_t is strongly convex with modulus $\gamma_t + \lambda$. Consequently the Bregman divergence is bounded by

$$D_{h_t}(w^*, w_t) \geq \frac{\gamma_t + \lambda}{2} \|w^* - w_t\|_2^2 .$$

Together with Lemma 10, we have

$$\begin{aligned}
& \mathbf{E}\left[f(w_t) - f(w^*) + \frac{\gamma_{t+1}}{2}\|w^* - w_t\|_2^2\right] \\
= \; & \mathbf{E}\left[f(w_t) - f(w^*) + \frac{\gamma_t + \lambda}{2}\|w^* - w_t\|_2^2\right] \\
\leq \; & \mathbf{E}\left[f(w_t)\right] - f(w^*) + \mathbf{E}\left[D_{h_t}(w^*, w_t)\right] \\
\leq \; & \mathbf{E}\left[D_{h_t}\left(w^*, w_{t-1}\right) - D_f\left(w^*; w_{t-1}\right)\right] + \frac{A^2}{b\gamma_t} \\
= \; & \mathbf{E}\left[D_{f_{I_t}}\left(w^*, w_{t-1}\right) - D_f\left(w^*; w_{t-1}\right)\right] \\
& + \frac{\gamma_t}{2}\mathbf{E}\left[\|w^* - w_{t-1}\|_2^2\right] + \frac{A^2}{b\gamma_t} \\
= \; & \frac{\gamma_t}{2}\mathbf{E}\left[\|w^* - w_{t-1}\|_2^2\right] + \frac{A^2}{b\gamma_t}
\end{aligned}$$

Here the first equality follows from the definition of γ_t; the second equality follows from the definition of h_t and simple algebra; the third equality uses the fact that we are drawing I_t independently. Hence we have

$$\mathbf{E}_{I_t|w_{t-1}} D_{f_{I_t}}(w^*, w_{t-1}) = D_f(w^*; w_{t-1}).$$

Summing over $t = 1, \ldots, T$, we obtain the desired bound. ■

Chapter 8

AdaDelay: Delay Adaptive Stochastic Optimization

8.1 Introduction

In this chapter we continue studying stochastic gradient descent for distributed optimization. Recall that the objective function to optimize is as follows (see also (7.1)):

$$w_* = \operatorname*{argmin}_{w \in \Omega} f(w), \quad \text{where } f(w) = \frac{1}{n} \sum_{i=1}^{n} f_i(w). \tag{8.1}$$

Instead of using synchronous iterations as studied in Chapter 7, in this chapter, we focus on asynchronous SGD and study how to improve the convergence by taking into account the actual delay caused by asynchronous updates.

Our work is motivated by the need of precise modeling the delay properties of real-world cloud services, which behavior are quite different from what one may observe with small clusters. In particular, computing resources in the cloud are shared by many users who run very different types tasks. Compared to an environment where resources are shared by a small number of individuals, the environment of cloud computing will inevitably be more diverse in terms of availability of key resources such as CPUs, disks, and network bandwidth, which all contribute to the delays in the computation process. Therefore, it is of great value to both service providers and end users of large-scale cloud services to be able to accommodate variable delays.

In light of this background, we investigate delay sensitive asynchronous SGD. In particular, instead of just using global "bounded delay" arguments as in Chapter 6, which can be too pessimistic, we aim to adapt the computation to the observed delays.

Contributions. A promising and intuitive approach to exploit the actual observed delay is as follows. At the early stage of the iterations, the server updates model parameters whenever its receives a gradient from any machine, with a weight inversely proportional to the actual delay observed. Towards the end, to reduce the bias caused by the initial aggressive steps, the server takes larger update steps upon receiving gradients from machines that send infrequent update, and takes smaller steps when the updates are from machines that update very frequently.

In this chapter, we propose and analyze a new asynchronous SGD algorithm, named *AdaDelay* (**Ada**ptive **Delay**), which uses step sizes that depend on the actual delays observed. It requires a more intricate convergence analysis for two reasons. First, the step sizes and are no longer guaranteed to be monotonically decreasing. Second, the residuals that measure progress are not independent over iterations as they are coupled by the random variable of delays.

The computation framework that we use to implement the algorithm is the Parameter Server (for more details see Chapter 3), where a central server maintains the global parameter, and the worker nodes compute stochastic gradients using their share of the data and communicate back to the central server to update the model. We validate our theoretical framework with experiments on large-scale data sets. The experiments reveal that our assumptions of network delay lead to a reasonable approximation of the observed delays in practice. In the regime of large delays, using delay sensitive steps is very helpful for obtaining fast convergence.

Related Work. Of particular relevance to our paper is the recent work on delay adaptive gradient scaling in an AdaGrad like framework [98], which claims substantial improvements under specialized settings over [50], a work that exploits data sparsity in a distributed asynchronous setting. Our experiments confirm the claims in [98] that their best learning rate is insensitive to the maximum delays. However, in our experience the method of [98] overly smooths the optimization path, which can have adverse effects on real-world data (see Section 8.3). To our knowledge, all previous works on asynchronous SGD assume monotonically diminishing step-sizes. Our analysis shows that rather than using worst case delay bounds, using exact delays to control step sizes can be remarkably beneficial in realistic settings, such as when there are stragglers that slow down progress for all the machines in a worst-case delay model.

Algorithmically, the work [4] is the one most related to ours; the authors of [4] consider using delay information to adjust the step size. However, the most important difference is that they only use the worst possible delays which again might be too conservative. The work in [79] investigates two variants of update schemes, both of which occur with delay, yet it does not exploit the actual delays either. There are some other interesting works studying specific scenarios, for example, [50], which focuses on the sparse data. However, our framework is more general and thus capable of covering more applications.

We build on the groundwork laid by [4, 103]; like them, we also consider optimizing (8.1) under a delayed gradient model.

Notation. We denote a random delay at time t by τ_t, denote step sizes by $\eta(t, \tau_t)$, and denote delayed gradients by $g(t-\tau_t)$. For a differentiable convex function h, the corresponding *Bregman divergence* is defined as $D_h(x, y) := h(x) - h(y) - \langle \nabla h(y),\, x - y \rangle$. We also interchangeably use x_t and $x(t)$ when it is clear in the context.

8.2 AdaDelay Algorithm

8.2.1 Model Assumptions

To simplify the exposition of our key ideas, we consider only smooth objective functions. Straightforward, albeit laborious extensions are possible to non-smooth functions, strongly convex costs, mirror descent versions, and proximal splitting versions. Following [4], we make the following standard assumptions on the objective function $f(x)$:

Assumption 11 (Lipschitz gradients) *The function f has locally L-Lipschitz gradients, namely*

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|, \qquad \forall\ x, y \in \Omega.$$

Assumption 12 (Bounded variance) *There exists a constant $\sigma > 0$ such that*

$$\mathbf{E}_\xi[\|\nabla f(x) - \nabla F(x;\xi)\|^2] \leq \sigma^2, \qquad \forall\, x \in \Omega.$$

Assumption 13 (Compact domain) *Let x^* be a global minimum of the function $f(x)$. Then,*

$$\max_{x\in\Omega} \|x - x^*\| \leq R.$$

Assumption 14 (Bounded Gradient) *Let $\forall\ x \in \Omega$. Then,*

$$\|\nabla f(x)\| \leq G.$$

These assumptions are typically reasonable for machine learning problems where the value of data examples are bounded. For instance, logistic-regression losses and least-squares costs all satisfy the assumptions.

We consider the following two delay models:

Uniform delay model. In this model, we assume that τ_t, the delay at time t, follows i.i.d. uniform distribution $U(\{0, 2\bar{\tau}\})$. This is a reasonable approximation of observed delays after a random start-up time of the network.

Scaled delay model. We assume that for every t, there is a $\theta_t \in (0, 1)$ such that $\tau_t < \theta_t t$. Moreover, assume that $\mathbf{E}[\tau_t] = \bar{\tau}_t$ and $\mathbf{E}[\tau_t^2] = B_t^2$, where $\bar{\tau}_t$ and B_t are constants that do not grow with t. This delay model includes all delay processes with bounded first and second moments, and is more general than the uniform model.

Our analysis seems general enough to cover many other delay distributions by combining the above two delay models. For example, the Gaussian model (where τ_t obeys a Gaussian distribution but its support must be truncated as $t > 0$) may be seen as a combination of the following arguments: when $t \geq C$ (a suitable constant), the Gaussian assumption indicates $\tau_t < \theta t$, which falls under our second delay model; when $0 \leq t \leq C$, our proof technique with bounded support (same as uniform model) applies.

8.2.2 Algorithm

Under the above two delay models, we consider the following projected stochastic gradient iteration:

$$w_{t+1} = \operatorname*{proj}_{\Omega}\left[w_t - \eta_t(\tau_t) g_{t-\tau_t}\right], \qquad t = 1, 2, \ldots, \tag{8.2}$$

where $g_{t-\tau_t}$ is the gradient that the server received at time t with an observed delay τ_t, and the learning rate $\eta_t(\tau_t)$ at iteration t is sensitive to the actual delay τ_t observed. Iteration (8.2) generates a sequence $\{w_t\}_{t\geq 1}$, and the server maintains the averaged iterate

$$\bar{w}(T) := \frac{1}{T}\sum_{t=1}^{T} w_{t+1}. \tag{8.3}$$

8.2.3 Convergence Analysis

Next, we analyze the convergence rate of the iterative algorithm in (8.2). The learning rate we use is of the form

$$\eta_t(\tau_t) = (L + \alpha_t(\tau_t))^{-1},$$

where the step offsets $\alpha_t(\tau_t)$ are chosen to be sensitive to the actual delay of the incoming gradients. We typically use

$$\alpha_t(\tau_t) = c\sqrt{t + \tau_t}, \tag{8.4}$$

for some constant c. For clarity of presentation we first let c be independent of t. Later we also consider time-varying c_t (see Corollary 18). The constant c serves to trade off between the effect of the noise variance σ and the effect of the radius bound R on the error bound. Note that if there is no delay, namely $\tau_t = 0$, it reduces to the standard synchronous SGD.

Our convergence analysis builds upon that of [4], whereas the key difference is that our step size $\eta_t(\tau_t)$ depends on the actual delay τ_t observed. These delay dependent step sizes necessitate more intricate analysis. The primary complexity arises from $\eta_t(\tau_t)$ being dependent of the actual delay τ_t, and thus no longer monotonically decreasing, as is typically assumed in most convergence analyses of SGD.

Theorem 15 *Let w_t be generated according to* (8.2)*. In the uniform delay model, we have*

$$\begin{aligned}\mathbf{E}\left[\sum_{t=1}^{T}\big(f(w_{t+1}) - f(w^*)\big)\right] \leq &\left(\sqrt{2}cR^2\bar{\tau} + \frac{\sigma^2}{c}\right)\sqrt{T} + \frac{LG^2(4\bar{\tau}+3)(\bar{\tau}+1)}{6c^2}\log T \\ &+ \tfrac{1}{2}(L+c)R^2 + \bar{\tau}GR + \frac{LG^2\bar{\tau}(\bar{\tau}+1)(2\bar{\tau}+1)^2}{6(L^2+c^2)}.\end{aligned}$$

In the scaled delay model, we have

$$\begin{aligned}\mathbf{E}\left[\sum_{t=1}^{T}\big(f(w_{t+1}) - f(w^*)\big)\right] \leq &\frac{\sigma^2}{c}\sqrt{T} + \frac{1}{2}cR^2\sum_{t=2}^{T}\frac{\bar{\tau}_t+1}{\sqrt{2t-1}} + GR\left[1 + \sum_{t=1}^{T-1}\frac{B_t^2}{(T-t)^2}\right] \\ &+ G^2\sum_{t=1}^{T}\frac{B_t^2 + 1 + \bar{\tau}_t}{L^2 + c^2(1-\theta_t)t} + \frac{1}{2}R^2(L+c).\end{aligned}$$

The detailed proof of Theorem 15 is technical and long. For the coherence of the thesis, we omit it here and direct the interested reader to the full version of our paper [127].

Theorem 15 has several implications. Corollaries 16 and 17 indicate that both our delay models share a similar convergence rate, while Corollary 18 shows that such results still hold even when we replace the constant c with a bounded sequence $\{c_t\}$. Finally, Corollary 19 mentions a simple variant that uses $\eta_t = c_t(t + \tau_t)^\eta$ for $0 < \eta < 1$, and it also highlights the fact that for $\eta = 0.5$, our algorithm achieves the best theoretical convergence.

Corollary 16 *Under the uniform delay model, we have*

$$\mathbf{E}[f(\bar{w}_T) - f^*] = \mathcal{O}\left(D_1\frac{\sqrt{T}}{T} + D_2\frac{\log T}{T} + D_3\frac{1}{T}\right),$$

where the constants

$$D_1 = \sqrt{2}cR^2\bar{\tau}+\frac{\sigma^2}{c}, D_2 = \frac{LG^2(4\bar{\tau}+3)(\bar{\tau}+1)}{6c^2}, D_3 = \tfrac{1}{2}(L+c)R^2+\bar{\tau}GR+\frac{LG^2\bar{\tau}(\bar{\tau}+1)(2\bar{\tau}+1)^2}{6(L^2+c^2)}.$$

Corollary 17 *Under the scaled delay model, let $\bar{\tau}_t = \tau$, $\theta_t = \theta$, and $B_t = B$ for all t. We have that*

$$\mathbf{E}[f(\bar{w}_T) - f^*] = \mathcal{O}\left(D_4\frac{\sqrt{T}}{T} + D_5\frac{\log\left(1+\frac{c^2(1-\theta)T}{L^2}\right)}{T} + D_6\frac{1}{T}\right),$$

where the constants

$$D_4 = \left[\frac{1}{\sqrt{2}}cR^2(\bar{\tau}+1)+\frac{\sigma^2}{c}\right], D_5 = \frac{G^2(B^2+\tau+1)}{c^2(1-\theta)}, D_6 = \tfrac{1}{2}(L+c)R^2 + GR\left(1+\frac{\pi^2B^2}{6}\right).$$

Corollary 18 *If $\alpha_t(\tau_t) = c_t\sqrt{t+\tau_t}$ with $0 < M_1 \leq c_t \leq M_2$, then the conclusion of Theorem 15, Corollary 16 and 17 still hold, except that the term c is replaced by M_2 and $\frac{1}{c}$ by $\frac{1}{M_1}$.*

If we wish to use step size offsets $\alpha_t(\tau_t) = c_t(t + \tau_t)^\gamma$ where $0 < \gamma < 1$, we get a result of the following form.

Corollary 19 *Let $\alpha_t(\tau_t) = c_t(t + \tau_t)^\gamma$ with $0 < M_1 \leq c_t \leq M_2$ and $0 < \gamma < 1$. Then, there exists a constant D_7 such that*

$$\mathbf{E}[f(\bar{w}_T) - f^*] = \mathcal{O}\left(\frac{D_7}{T^{\min(\gamma,1-\gamma)}}\right).$$

8.3 Experiments

8.3.1 Setup

We evaluate the performance of the proposed algorithm AdaDelay in a distributed computing environment.

We compare AdaDelay with two related methods called AsyncAdaGrad [4] and AdaptiveRevision [98]. Let $[\eta_t(\tau_t)]_j$ denote the j-th coordinate of $\eta_t(\tau_t)$, and let $[g_t(\tau_t)]_j$ denote the delayed gradient on the j-th feature. AsyncAdaGrad uses a scaled learning rate given by

$$[\eta_t(\tau_t)]_j = \left(\sum_{i=1}^{t} [g_i(\tau_i)]_j^2 \right)^{1/2}.$$

AdaptiveRevision takes into account the actual delays by considering $\left[g_t^{\text{bak}}(\tau_t)\right]_j = \sum_{i=t-1}^{t-\tau} [g_i(\tau_i)]_j$, and uses a non-decreasing learning rate given by

$$[\eta_t(\tau_t)]_j = \left(\sum_{i=1}^{t} [g_i(\tau_i)]_j^2 + 2[g_t(\tau_t)]_j \left[g_t^{\text{bak}}(\tau_t)\right]_j \right)^{1/2}.$$

The key novelty of our algorithm AdaDelay is the delay-dependent learning rate given by

$$\eta_t(\tau_t) = \alpha_0 (L + \alpha_t(\tau_t))^{-1}.$$

We follow the common practice to fix $L = 1$ while searching for the best α_0. Similar to AsyncAdaGrad and AdaptiveRevision, we use a scaled learning rate in AdaDelay to better model the nonuniform sparsity of the data set (this step size choice falls within the purview of Corollary 18). In other words, we set

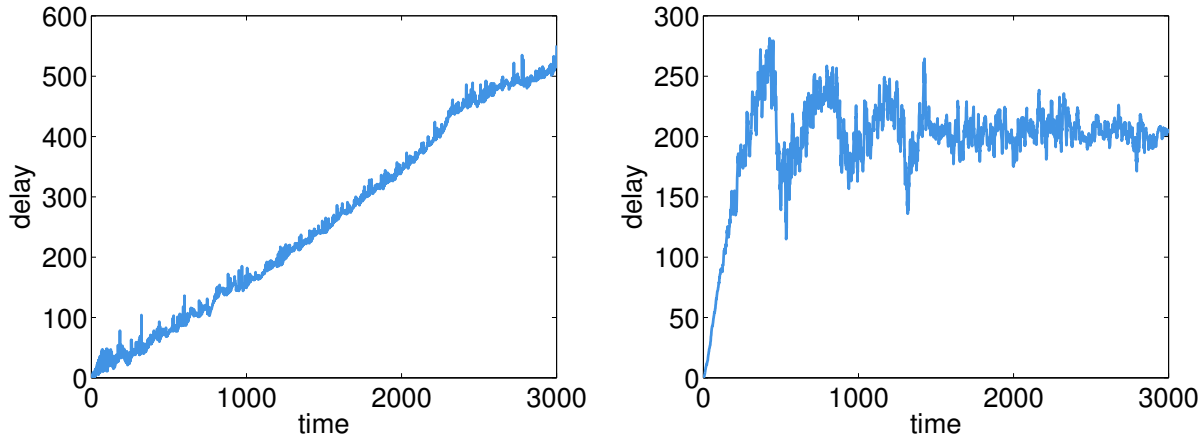
$$[\alpha_t(\tau_t)]_j = c_j \sqrt{t + \tau_t},$$

where $c_j = (\frac{1}{t} \sum_{i=1}^{t} \frac{i}{i+\tau_i} [g_i(\tau_i)]_j^2)^{1/2}$ serves to average the weighted delayed gradients.

We test the algorithms for the objective function of logistic regression on the dataset CTRa and Criteo. For Criteo, the first 8 days are used for training while the following 2 days are used for validation. For CTRa, we sampled another 20 millions examples to test the trained model. All the experiments were run using the campus cluster CampusA.

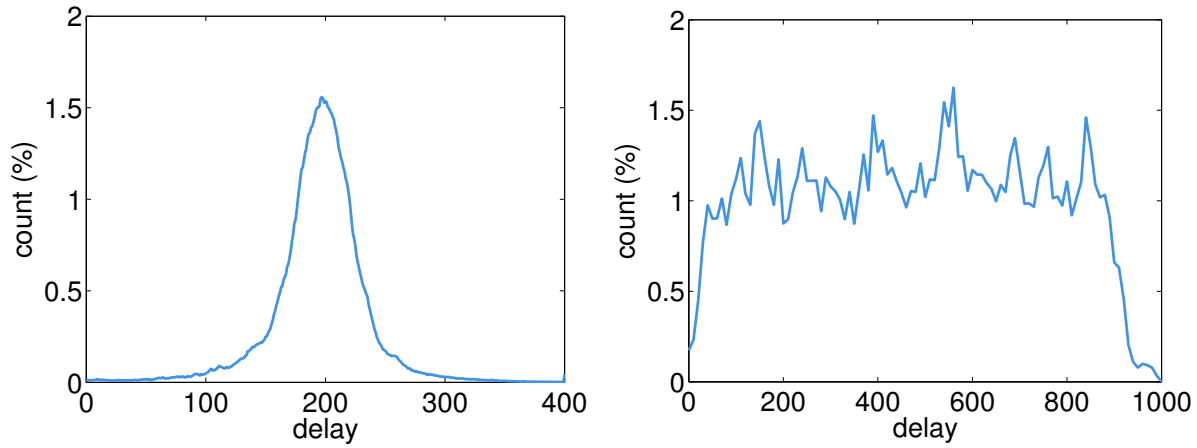
We implement the three methods in the parameter server framework introduced in Chapter 3. In each iteration, a worker node first reads a minibatch of data from a distributed file system, pulls the required working set of parameters from the server nodes, compute the gradients on the minibatch, and then pushes the gradients to the server nodes. Let $t(k, i)$ denote the time when worker k pulls the weight for minibatch i, and let $t'(k, i)$ denote the time when the server nodes update the weight upon receiving the gradients on this minibatch. The actual delay caused by processing this minibatch is thus given by $\tau_t = t'(k, i) - t(k, i)$.

The server nodes maintain the model parameters. For each feature, both AsyncAdaGrad and AdaDelay need to store the weight and the accumulated gradient to compute the learning rate, while AdaptiveRevision needs two more entries, including g_j^{bck} for each feature j. If we send g_j^{bck} over the network following the method in [98], the total network communication increases by 50%. Instead, we store g_j^{bck} at the server nodes while processing this minibatch, which incurs no network communication overhead at the cost of additional memory consumption.



(a) Dataset Criteo with 1,600 workers (b) Dataset CTRa with 400 workers

Figure 8.1: The first 3,000 observed delays on one server node.



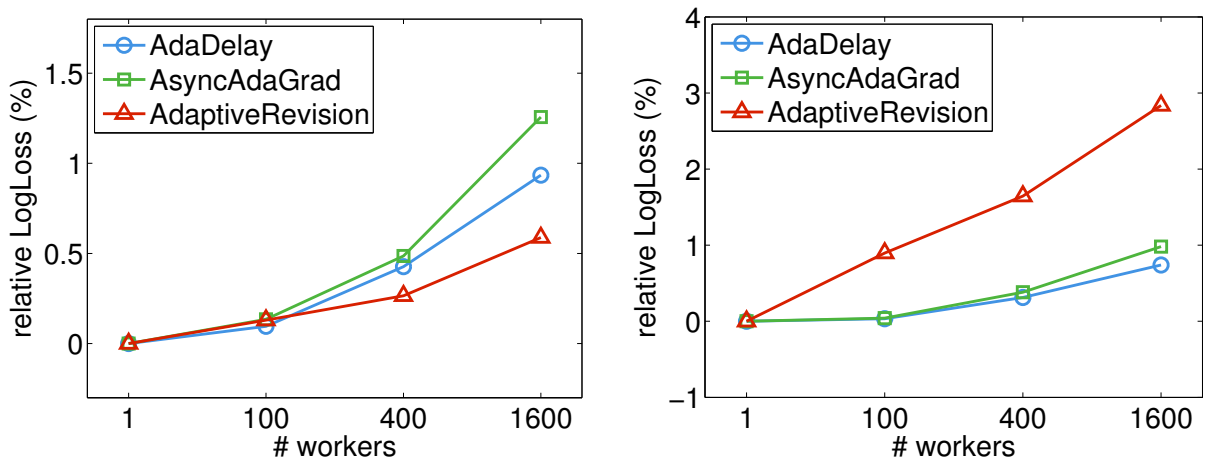
(a) Dataset Criteo with 1,600 workers (b) Dataset CTRa with 400 workers

Figure 8.2: Histgram of all observed delays

8.3.2 Results

Delay observed Figure 8.1 and 8.2 visualize the actual delay observed at server nodes for AdaDelay. We observe that delay τ_t is roughly linear in t at the early stage of the training process, with different slope for different tasks. For example, the slope is around 0.2 for the Criteo data set with 1,600 workers, and it increases to around 1 for the CTRa data set with 400 workers. At later stage of the training process, after the delay increases to around half of the number of workers, it roughly follows a unimodal distribution.

Loss and AUC Next, we present the comparison results of these three algorithms by varying the number of workers. Following [98] we use the online LogLoss as the criterion. That is, given example with feature vector d and label $y \in \{+1, -1\}$, we calculate the loss function



(a) Dataset Criteo (b) Dataset CTRa

Figure 8.3: Relative (% worsening) of online LogLoss as function of maximal delays (lower is better).

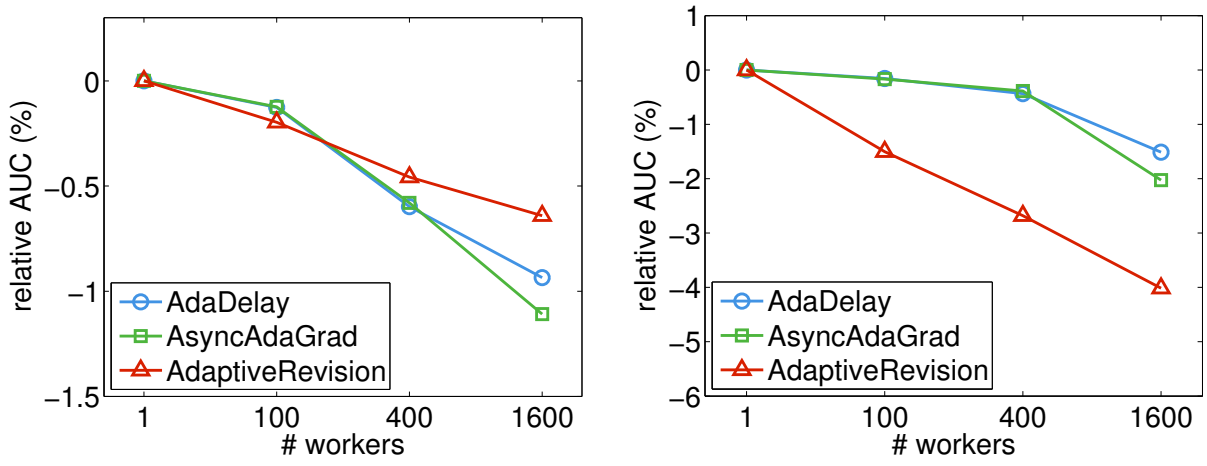
$f(x,(d,y)) = \log(1+\exp(-y\langle d,x\rangle))$ before updating x using (y,d). Similar to [98], we report the average LogLoss over the second half of the training data to ignore the possible large values when starting training.

Figure 8.3 reports the relative change in online LogLoss for the three algorithms compared (smaller value is better). It is seen that on the Criteo dataset, AdaDelay performs better than AsyncAdaGrad, though AdaptiveRevision is slightly better than both. However, for the larger CTR2 dataset, both AdaDelay and AsyncAdaGrad are substantially better than AdaptiveRevision. The reason why it differs from [98] is probably due to the datasets we used are 1000 times larger than the ones reported by [98], and we evaluated the algorithms in a distributed environment rather than a simulated setting where a large minibatch size is necessary for the former. However, as reported [98], we also observed that AdaptiveRevision's best learning rate is insensitive to the number of workers.

AdaDelay seems to have a tiny edge over AsyncAdaGrad, and as predicted by our theory, this edge grows much bigger when there are large delays (e.g., due to stragglers)—we report on this in greater detail in next paragraph.

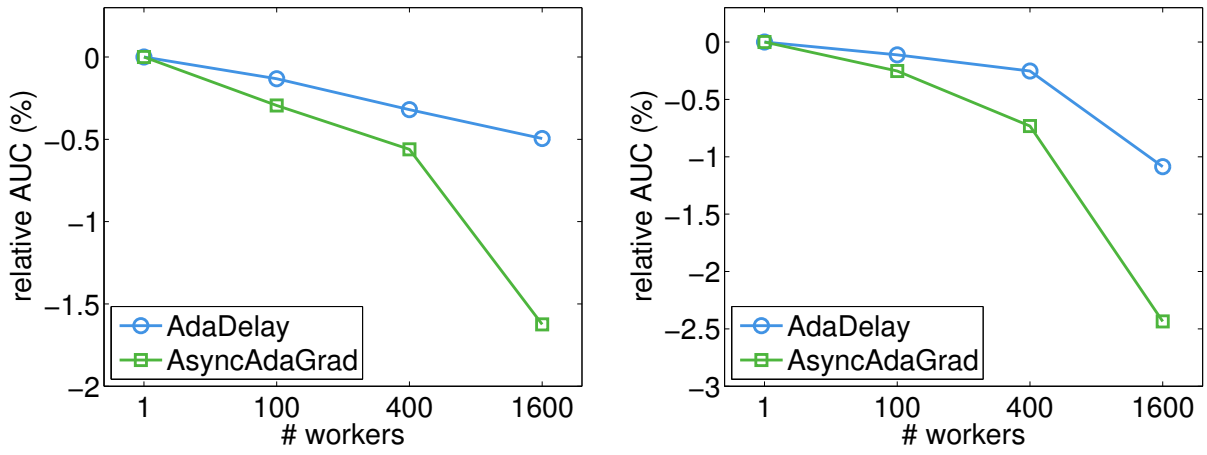
Besides the LogLoss, AUC is another important merit for computational advertising, which measures the ranking ability of the model and often 1% difference is significant for click-through rate estimation. We made a separate validation dataset for calculating the AUC, and shown the results on Figure 8.4. As can be seen, the test AUC results are consistent with the online LogLoss.

Stragglers Previous experiments indicate that AdaDelay improves upon AsyncAdaGrad when a large number of workers (greater than 400) is used, which means the delay adaptive learning rate takes effect when the delay can be large. To further investigate this phenomenon, we simulated an overloaded cluster where several stragglers may produce large delays; we do this by slowing down half of the workers by a random factor in $[1,4]$ when computing gradients. The



(a) Dataset Criteo (b) Dataset CTRa

Figure 8.4: Relative test AUC (higher is better) as function of maximal delays.



(a) Dataset Criteo (b) Dataset CTRa

Figure 8.5: Relative test AUC (higher is better) as function of maximal delays with the existence of stragglers.

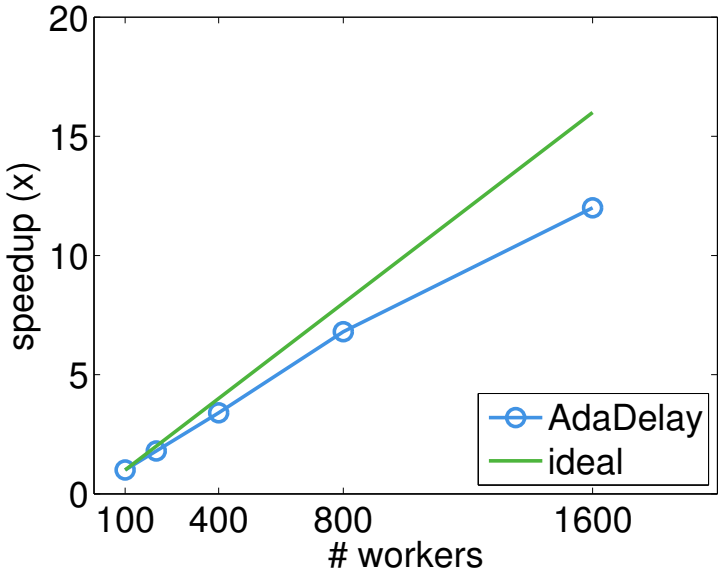


Figure 8.6: The speedup of AdaDelay. The results of AsyncAdaGrad and AdaptiveRevision are almost identical to AdaDelay and therefore omitted.

	AdaDelay	AsyncAdaGrad	AdaptiveRevision
Criteo	24 GB	24 GB	55 GB
CTRa	97 GB	97 GB	200 GB

Table 8.1: Total memory used by server nodes.

test AUC are shown in Figure 8.5[1]. As can be seen, AdaDelay consistently outperforms AsyncAdaGrad, which shows that adaptive modeling of the actual delay is better than using a constant worst case delay when the variance of the delays is large.

Scalability Finally we report the system performance. We first present the speedup from 1 machine to 16 machines, where each machine runs 100 workers. We observed a near linear speedup of AdaDelay, which is shown in Figure 8.6. The main reason is due to the asynchronous updating which removes the dependencies between worker nodes. In addition, using multiple workers within a machine can fully utilize the computational resources by hiding the overhead of reading data and communicating the parameters. The results of AsyncAdaGrad and AdaptiveRevision are similar to AdaDelay because their computational workloads are identical except for parameter updating, which affects the overall system performance little.

In the parameter server framework, worker nodes only need to cache one or a few data mini-batches. Most memory is used by the server nodes to store the model. We summarize the server memory usage for the three algorithms compared in Table 8.1.

As expected, AdaDelay and AsyncAdaGrad have similar memory consumption because the extra storage needed by AdaDelay to track and compute the incurred delays τ_t is tiny. However

[1] As before, the results on online LogLoss are similar to the test AUC and therefore omitted.

AdaptiveRevision doubles memory usage, because of the extra entries that it needs for each feature, and because of the cached delayed gradient g^{bak}.

Chapter 9

Parsa: Data Partition via Submodular Approximation

9.1 Introduction

In this chapter, we attempt to address one question that is fundamental to applying today's loosely-coupled "scale-out" cluster computing techniques to important classes of machine learning applications: how to spread data and parameters around a cluster of machines in order to achieve efficient processing?

In our Parameter Server (see Chapter 3), both data and parameters are distributed over the machines to fit the computation power and storage capacity of the nodes. Many other applications and frameworks also face the issue of data and parameter distribution. For instance, in very large scale graph factorization [7], the challenge is to partition a natural graph, so that the memory required to store the local states and to cache adjacent variables is bounded within the capacity of a machine. Similar constraints appear in GraphLab [59, 90] where vertex-specific updates are carried out while other variables need to be synchronized between machines. Likewise, in distributed graphical model inference with latent variables [6, 123], state variables must be distributed and synchronized efficiently between machines.

Shared parameters are synchronized via communication network in a scale-out cluster. The sheer size of model parameters and the iterative nature of machine learning algorithms often produce huge amount of network traffic. Figure 9.1 shows that, in a text classification application, if we randomly assign examples (documents) to machines, the total amount of network traffic can be 10 times higher than the size of training data. Specifically, more than 1 Terabyte of data are communicated for 72 Gigabyte of training data. Given that network bandwidth is typically much smaller than the bandwidth of accessing local memory, the sheer amount of traffic potentially becomes a performance bottleneck for many large scale machine learning applications.

In particular, we face three key challenges in order to achieve scalability:

- Memory footprint (RAM) is limited, and hence the amount of storage per machine available for processing and caching model parameters is often quite constrained compared to

[2]We assume 100 machines and the optimization algorithm running 100 iterations until convergence. The sparse first-order gradient are communicated in each iteration.

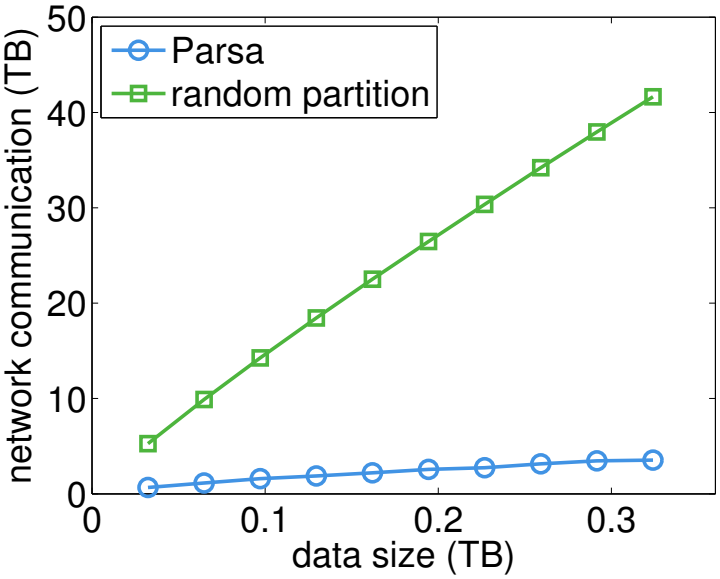


Figure 9.1: The amount of network communication versus the size of data in a real text classification dataset for random partition[2].

the size of the model.

- Network bandwidth and latency are roughly 100 times slower than accessing local memory, thus one would like to minimize the amount of synchronization between machines.
- The computing power per machine is also limited, which demands a well-balanced task distribution over machines.

One useful observation is that a lot of times large scale data are extremely sparse. For example, documents usually contain only a small fraction of distinct words; one person usually has only a few friends compared to the total number of people in a social graph; there are only few vertices in a PubSub network with very high degree (e.g., Bieber, Gaga and Kutcher on Twitter). Such non-uniformity and sparsity is both a boon and a challenge —we are a nontrivial problem of partitioning the model for distributed computing, and simple random partitioning typically leads to poor performance.

Previous works Graph partitioning has attracted a lot of research interests in scientific computing [28, 47, 73], scaling out large-scale computations [20, 59, 60, 141, 147, 155], graph database [39, 122, 144], search and social network analysis [7, 112, 141], and streaming processing [105, 130, 131, 140]. Most of the previous works, such as the well-known package METIS [73], were concerned with edge cuts. Only a few of them solved the vertex cut problem, which is closely related to this paper, to directly minimize the network traffic. PaToH [28] and Zoltan [47] used multilevel partitioning algorithms related to METIS, while PowerGraph [59] adopted a greedy algorithm. Very recently [20] studied the relation between edge cut and vertex cut.

Our contribution Even though partitioning problems are often NP hard [131], for the practical importance of this problem, it is worth seeking scalable solutions that outperform random partitioning. We model the data and parameter placement problem as a graph partitioning problem, and we propose *Parsa*, a PARallel Submodular Approximation algorithm, for solving this problem. We also provide a fast and practical implementation using an efficient vertex selection data structure. We propose a novel method based on submodular approximation to solve the vertex-cut partitioning problem, and provide theoretical analysis for the partition quality. Our implementation has runtime $\mathcal{O}(k|E|)$, where k is the number of partitions and $|E|$ denotes the number of edges in the graph. This is nontrivial compared to the straightforward implementation which has runtime $\mathcal{O}(k|E|^2)$. We further describe technologies including sampling, initialization and parallelization to improve the partitioning quality and efficiency. On several large scale datasets, we show that the proposed algorithm outperforms the state-of-the-art methods, including METIS [73], PaToH [28] and Zoltan [47], in terms of both partition quality and speed. Moreover, the proposed method can significantly accelerate Parameter Server, when dealing with hundreds gigabytes of data and billions of model parameters.

9.2 Problem Formulation

Many inference problems in machine learning have graph-structured dependencies. For instance, we can rewrite the objective of risk minimization ((1.1) in Chapter 1) as

$$\underset{w}{\text{minimize}} \sum_{i=1}^{n} f(x_i, y_i, w). \tag{9.1}$$

In a lot of applications, data and model parameters are often correlated via the nonzero and sparse terms in x_i. For example, in spam filtering for email services, these terms correspond to words and attributes in emails; while in computational advertising, they can model typical words in ads and user behavior patterns.

For inference problems in undirected graphical models [18, 76], the correlations between random variables can be encoded by clique potentials $\psi_C(w_C)$, each of which only depends on the subset of variables w_C belonging to a clique C. The applications, ranging from message passing, optimization, to sampling, all require extensive manipulation of the parameters involved in a given clique w_C. That is, one needs to solve problems of the form

$$\underset{w}{\text{minimize}} \sum_{C \in \mathcal{C}} \psi_C(w_C), \tag{9.2}$$

where $\mathcal{C}$ is a set of cliques of variables.

Similar problems also occur in the context of inference on natural graphs [7, 9, 59]. Again, we have sets of interacting parameters which can be represented by vertices on a graph. Manipulating a vertex affects all of its neighbors in terms of computation.

In the rest of this section, we first show how the parameter dependencies can be modeled using bipartite graphs. Then we formulate the parameter and data placement problem as a graph partition problem.

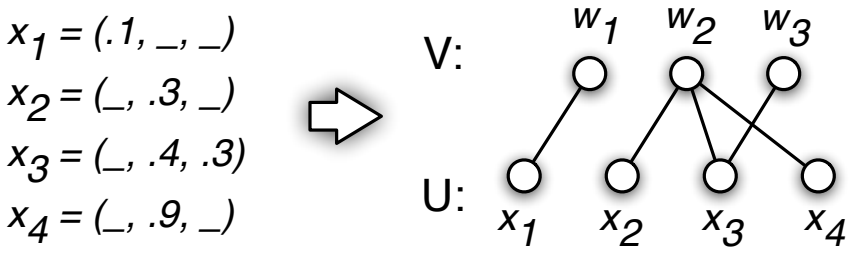


Figure 9.2: The dependencies are modeled as a bipartite graph.

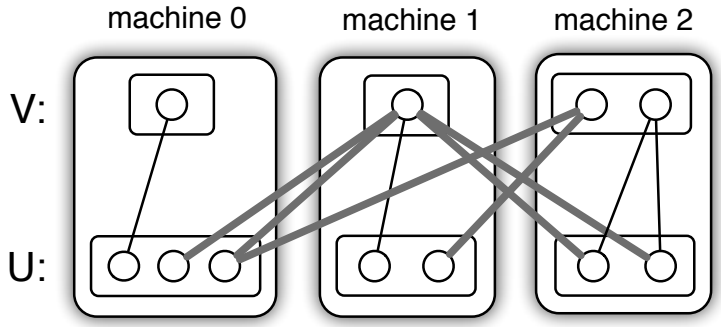


Figure 9.3: Each machine is assigned with a server and a worker, and gets part of the vertex set U and V. The inter-machine dependencies (edges) are highlighted and the communication costs for these three machines are 1, 3, and 3, respectively. Note that moving the 3rd vertex in V to either machine 0 or machine 1 can reduce the cost.

Bipartite Graphs The dependencies in the inference problems can be modeled with a bipartite graph. Let $G(U, V, E)$ denote a bipartite graph, where U and V are the two vertex sets, and E is the edge set with entries $(u, v) \in E$ for $u \in U$ and $v \in V$. Figure 9.2 illustrates how the dependencies in the problem of risk minimization can be modeled with a bipartite graph. Here U correspond to data examples $\{u_i = (x_i, y_i)\}_{i=1}^m$ and V correspond to the model parameters w. There is an edge $(u_i, v_j) \in E$ if and only if the j-th entry of x_i is nonzero. Therefore, $\{w_j : (u_i, v_j) \in E\}$ is the working set of parameters which are needed for evaluating the loss function $l(x_i, y_i, w)$. In other applications such as graphical models and nature graph inference where there is a natural undirected graph $G(V, E)$, we can easily convert the graphs into bipartite graphs $G(U', V, E')$. One way is to make U' a copy of the original vertex set V, and include to the edge set $(u_i, v_j) \in E'$ if $(v_i, v_j) \in E$. Alternatively we can define U' to be the set of all the cliques on the original graph G, and include the edge $(u_i, v_j) \in E'$ if and only if the vertex v_j is contained in the clique u_i.

Graph Partition In distributed inference, we need to partition the problem and assign the computational tasks to m machines. Next, we show that this can be formulated as a problem of partitioning the dependency graph.

Take Parameter Server as an example. Without loss of generality, assume that there are m workers and m servers, and that each machine is associated with one worker and one server. Note

that we can combine multiple workers or servers assigned to the same machine into a single node without affecting the following analysis. To partition the graph, we divide the vertex set V for the parameters into m non-overlapping subsets, namely $V = \bigcup_{i=1}^{m} V_i$, and assign each subset to a server. Similarly, we partition the vertex set U for the data into m non-overlapping subsets, namely $U = \bigcup_{i=1}^{m} U_i$, and assign them to the workers. Figure 9.3 shows an example for $m = 3$.

Next, we formalize the three goals we want to achieve with an efficient graph partition.

Balance the computational load We assume that each example u_i incurs roughly the same amount of computational workload. In order to ensure that each machine is assigned approximately the same workload, we try to keep $\max_i |U_i|$ small:

$$\text{minimize} \quad \max_i |U_i|. \tag{9.3}$$

Satisfy the memory constraints Inference algorithms need to frequently access the model parameters, which are stored in the limited local memory of the workers. Let $\mathcal{N}(u_i) = \{v_j : (u_i, v_j) \in E\}$ denote the neighbor set of the (data) vertex u_i. We define $\mathcal{N}(U_i) = \bigcup_{u \in U_i} \mathcal{N}(u)$ to be the working set of the parameters that worker i need for its computation. Assume that each parameter v_j has the same storage cost, we try to limit each worker's memory footprint by:

$$\text{minimize} \quad \max_i |\mathcal{N}(U_i)|. \tag{9.4}$$

In the following, we define function $q(U) := |\mathcal{N}(U)|$.

Minimize the communication cost The communication cost of worker i scales with $|\mathcal{N}(U_i)|$. To further reduce this communication cost, we configure server i at the same machine where worker i resides, so that the data communication between this worker and this server has a large bandwidth. The communication cost of worker i only scales with $(|\mathcal{N}(U_i)| - |\mathcal{N}(U_i) \backslash V_i|)$. Moreover, if the model parameter v_j is not used by worker i, server i does not need to store it. In other words, we enforce $V_i \subseteq \mathcal{N}(U_i)$ and thus the communication cost scales with $(|\mathcal{N}(U_i)| - |V_i|)$. The communication cost of server i scales with $(\sum_{j \neq i} |V_i \cap \mathcal{N}(U_j)|)$, which is incurred when other workers request parameters from this server. Overall, we try to reduce the maximal communication cost of each machine by

$$\text{minimize} \quad \max_i |\mathcal{N}(U_i)| - |V_i| + \sum_{j \neq i} |V_i \cap \mathcal{N}(U_j)|. \tag{9.5}$$

9.3 Algorithm

In this section we present Parsa, our algorithm for solving the graph partition problem. Note that the optimization problem in (9.4) equivalent to a m-way graph partition problem on vertex set U with vertex-cut as the merit, which has been shown to be NP-Complete [28]. Instead of trying to solve the three hard optimization problems in (9.3)-(9.5) exactly and simultaneously, the proposed algorithm provides approximate solutions. In particular, the algorithm Parsa proceeds in two steps: first, it partitions the vertex set U for the data by solving (9.3) and (9.4)

Algorithm 10 Partition U via submodular approximation

Input: Graph G, number of partitions m, maximal number of iterations T, residue θ, and improvement α
Output: partitions of $U = \bigcup_{i=1}^{m} U_i$
for $i = 1, \ldots, m$ **do**
    $U_i \leftarrow \emptyset$
    define $g_i(S) := f(S \cup U_i) - \alpha|S \cup U_i|$.
end for
for $t = 1, ..., T$ **do**
    if $|U| \leq m\theta$ **then break**
    find $i \leftarrow \text{argmin}_j |U_j|$
    draw $R \subseteq U$ by choosing $u \in U$ with probability $\frac{n}{|U|m}$
    if $|R| > 2T/m$ **then next**
    solve $S^* = \text{argmin}_{S \subseteq R} g_i(S)$
    if $g_i(S^*) \leq 0$ **then**
        $U_i \leftarrow U_i \cup S^*$ and $U \leftarrow U \setminus S^*$
    end if
end for
if $|U| > m\theta$ **then return** fail
evenly assign the remainder U to U_i

approximately, namely assigning the examples to worker nodes to balance the CPU load and minimize the memory footprint; then given the partition of U, it divides the vertex set V for the parameters by solving (9.5) exactly, namely dividing the parameters to server nodes to minimize the inter-machine communication cost.

9.3.1 Partition the data vertex set U

Note that $q(U) = |\mathcal{N}(U)|$ is a submodular function with respect to U. Although the problem of minimizing $\max_i q(U_i)$ is NP-Complete, there exist a few approximate algorithms. For example [133] proposed an algorithm with the guarantee that all U_i have approximately equal size. In Algorithm 10, we sketch our algorithm for solving (9.3) and (9.4). It is based on the method in [133]. The key difference is that we build up the sets U_i in an incremental way, which is essential for guaranteeing both partition quality and computation efficiency at a later stage.

Our algorithm proceeds in the following steps. Each time we choose the smallest subset U_i and find the a subset of vertices to add to it. More specifically, we draw a small set of candidate vertices R and select the best subset of R by solving the following optimization:

$$S^* = \arg\min_{T \subseteq R} g_i(T) := q(U_i \cup T) - \alpha|U_i \cup T|, \tag{9.6}$$

where the constant α is the minimum-increment weight. If the optimal solution S^* to the above optimization satisfies $q(U_i \cup S^*) < \alpha|U_i \cup S^*|$, i.e. the cost for increasing U_i is small enough, we assign S^* to the subset U_i.

The quality of the obtained partition of U is analyzed in the following theorem.

Theorem 20 *Assume that there exists some partitioning U_i^* that satisfies $\max_i q(U_i^*) \leq B$. Let $m > \theta = \sqrt{T/\log T}$, $c = (32\pi)^{-\frac{1}{2}}$, $\alpha = mB/\sqrt{T \log T}$ and $\tau = \frac{S^3}{c} \log \frac{1}{1-p}$. Algorithm 10 succeeds with probability at least p and it generates a feasible solution with a cost*

$$\max_i q(U_i) \leq 4B\sqrt{T/\log T}.$$

Proof. The proof closely follows that in [133].

For a given iteration, assume that U_1^* maximizes $|U_j^* \cap U|$ for all j. Define $\upsilon = |U_1^* \cap U|$. Since S^* is the optimal solution at the current iteration we have $g_i(S^*) \leq g_i(U_1^* \cap U) = f((U_1^* \cap U) \cup U_i) - \alpha\,|(U_1^* \cap U) \cup U_i|$. Further note that by monotonicity and submodularity $f((U_1^* \cap U) \cup U_i) \leq q(U_1^* \cap U) + q(U_i)$. Moreover, $U \cap U_i = \emptyset$ holds since U contains only the leftovers. Consequently $|(U_1^* \cap U) \cup U_i| = |U_1^* \cap U| + |U_i|$. Finally, the algorithm only increases the size of U_i whenever the cost is balanced. Hence $q(U_i) - \alpha|U_i| < 0$. Combining this yields

$$g_i(S^*) \leq g_i(U_1^* \cap U) \leq q(U_1^* \cap U) - \alpha|U_1^* \cap U| \leq B - \alpha\upsilon.$$

Using the Theorem 5.4 from the proof of [133] we know that $\upsilon \geq B/\alpha$ and therefore $g_i(S^*)$ happens with probability at least c/S^2. Hence the probability of removing at least one vertex from U within an iteration is greater than c/S^2.

Chernoff bounds show that after $\tau = -S^2/c \log(1-\delta)$ iterations the algorithm will terminate with probability at least p since the residual U is small, i.e., $|U| \leq m\theta$.

The algorithm will never select a U_i for augmentation unless $|U_i| \leq T/m$ (there would always be a smaller set). Moreover, the maximum increment at any given time is $2T/m$. Hence $|U_i| \leq 3T/m$ and therefore $q(U_i) \leq 3T\alpha/m$.

Finally, the contribution of the unassigned residual U is at most θB since each U_i is incremented by at most θ elements and since $q(u) \leq B$ for all $u \in U$. In summary, this yields $q(U_i) \leq 3T\alpha/m + B\theta = 4B\sqrt{T/\log T}$. ■

9.3.2 Partition the parameter vertex set V

Given a partition of the data vertex set U, to find an assignment of parameters in the vertex set V to the servers, we reformulate the problem in (9.5) as a convex integer programming problem with totally unimodular constraints. This is then solved by a sequential optimization algorithm that performs a sweep through the variables.

We define indicator variables $v_{ij} \in \{0, 1\}$ for $j = 1, \ldots, m$ to specify which server node maintains a particular model parameter corresponding to v_i. A straightforward constraint is $\sum_{j=1}^m v_{ij} = 1$. Also, let $u_{ij} \in \{0, 1\}$ denote whether $j \in \mathcal{N}(U_i)$. We rewrite (9.5) as a convex integer program:

$$\underset{v}{\text{minimize}} \;\; \max_i |\mathcal{N}(U_i)| + \sum_j v_{ij} \left[-1 + \sum_{l \neq i} u_{lj} \right] \tag{9.7a}$$

$$\text{subject to} \sum_j v_{ij} = 1, \quad v_{ij} \in \{0, 1\}, \quad \text{and } v_{ij} \leq u_{ij}. \tag{9.7b}$$

Algorithm 11 Partition V for given $\{U_i\}_{i=1}^m$

1: **Input:** the neighbor sets $\{\mathcal{N}(U_i)\}_{i=1}^m$
2: **Output:** partitions $V = \bigcup_{i=1}^m V_i$
3: **for** $i = 1, \dots m$ **do**
4: $V_i \leftarrow \emptyset$
5: $\text{cost}_i \leftarrow |\mathcal{N}(U_i)|$, which is the communication cost of machine i
6: **end for**
7: **for all** $j \in V$ **do**
8: $\xi \leftarrow \operatorname{argmin}_{i:u_{ij}\neq 0} \text{cost}_i$
9: $V_\xi \leftarrow V_\xi \cup \{j\}$
10: $\text{cost}_\xi \leftarrow \text{cost}_\xi - 1 + \sum_{i\neq\xi} u_{ij}$
11: **end for**

Here we exploit the fact that $\sum_j v_{ij}u_{lj} = |V_i \cap \mathcal{N}(U_l)|$ and that $\sum_{j=1}^m v_{ij} = |V_i|$. It turns out that the constraints satisfy the totally unimodularity conditions discussed in [66]. As a consequence, we can relax the constraint $v_{ij} \in \{0, 1\}$ to $v_{ij} \in [0, 1]$ to obtain a convex optimization problem. While the relaxed problem is still formidable with possibly billions of variables and thousands of partitions, it informs us that we have a unique minimum value in the objective. Furthermore, sequential minimization over one row of $v_{i\cdot}$ at a time leads to convergence to the optimal solution.

Algorithm 11 performs a single sweep over (9.7) to find a locally optimal assignment of one variable at a time. We observe that this method is sufficient for a near-optimal solution. Repeated sweeps over the assignment space are straightforward and continue improving the objective. Further note that we need not store the full neighbor sets in memory. Instead, we can perform the assignment in a streaming fashion.

9.4 Efficient Implementation

In this section, we discuss the details of our efficient implementation (Algorithm 12) of the proposed algorithm (Algorithm 10). In the rest of this section, we first show how to find the optimal S^* and how to sample the subset R in (9.6), then we address the issue of parallel implementation with our Parameter Server, and at last we comment on the issue of neighbor set initialization.

9.4.1 Find Solution to (9.6)

The most expensive operation in the inner loop of Algorithm 10 is line 12, which solves (9.6) to find S^*, the optimal set of vertices to add to the partition. Standard submodular minimization problems have runtime in the order of $\mathcal{O}(n^6)$ [107], which is impractical for large n. A key approximation in our implementation of Parsa is to add only a single vertex at a time instead of a set of vertices. In other word, instead of solving (9.6) exactly, given a vertex set R and partition i, it finds the vertex u^* as the solution to the following problem:

$$u^* = \operatorname*{argmin}_{u\in R} g_i(u) := |\mathcal{N}(\{u\} \cup U_i)| - \alpha|\{u\} \cup U_i|. \tag{9.8}$$

Algorithm 12 Parsa: PARallel Submodular Approximation

Input: Graph G, initial neighbor sets $\{S_i\}_{i=1}^m$, number of partitions m, max delay τ, initialization from a, number of subgraphs b.
Output: partitions $U = \bigcup_{i=1}^m U_i$

Scheduler:

divide G into b subgraphs
ask all workers to partition G with $(a, \tau, \text{true}, \text{false})$
ask all workers to partition G with $(b, \tau, \text{false}, \text{true})$

Server:

start with a part of $\{S_i\}_{i=1}^m$
if receiving a pull request **then**
    reply with the requested neighbor set $\{S_i\}_{i=1}^m$
end if
if receiving a push request containing $\{S_i^{\text{new}}\}_{i=1}^m$ **then**
    if initializing **then**
        $S_i \leftarrow S_i^{\text{new}}$ for $i = 1, \ldots, m$
    else
        $S_i \leftarrow S_i \cup S_i^{\text{new}}$ for $i = 1, \ldots, m$
    end if
end if

Worker:

receive parameters $(S, \tau, \text{initializing}, \text{output})$
for $t = 1, \ldots, T$ **do**
    load a subgraph G
    wait until all pushes before time $t - \tau$ finished
    pull the part of neighbor sets, $\{S_i\}_{i=1}^m$, that contained in V from the servers
    get partitions $\{U_i^{\text{new}}\}_{i=1}^m$ and updated neighbor sets $\{S_i^{\text{new}}\}_{i=1}^m$ using Algorithm 13
    if initialing = false and $t > 1$ **then**
        $S_i^{\text{new}} \leftarrow S_i^{\text{new}} \setminus S_i$ for all $i = 1, \ldots, m$
    end if
    push $\{S_i^w\}_{i=1}^m$ to servers
    if output **then** $U_i \leftarrow U_i \cup U_i^{\text{new}}$ for $i = 1, \ldots, m$
end for

An additional advantage of this approximation is that since we assign only one vertex to the smallest subset, the CPU load can be perfectly balanced for (9.3).

A naive way to solve (9.8) is to compute $g_i(u)$ for all vertices and then determine the optimal u^*. However, if the size of R is a constant fraction of the entire graph, this runtime $\mathcal{O}(|U||E|)$ still remains impractical for large graphs. We further speed up this computation by pre-computing and storing all vertex costs, and then creating a data structure which allows us to efficiently locate the lowest-cost vertex in the memory. Algorithm 13 sketches our method. The inputs to Algorithm 13 are the bipartite graph $G = (U, V, E)$, the number of partitions m, and m subsets

Algorithm 13 Partition the vertex set U

Input: graph $G(U, V, E)$, number of partitions m, and initial neighbor sets $\{S_i\}_{i=1}^m$
Output: partitioned $U = \bigcup_{i=1}^m U_i$ and updated neighbor sets which are equal to $\{S_i \cup \mathcal{N}(U_i)\}_{i=1}^m$
for $i = 1, \ldots, m$ **do**
    $U_i \leftarrow \emptyset$
    for all $u \in U$ **do**
        $\mathcal{A}_i(u) = |\mathcal{N}(u) \setminus S_i|$
    end for
    $\mathcal{A}_i$.min $:= \text{argmin}_{u \in U} \mathcal{A}_i(u)$ which always points to the lowest-cost vertex
end for
while $|U| > 0$ **do**
    pick partition $i \leftarrow \arg\min_j |S_j|$
    pick the lowest-cost vertex $u^* \leftarrow \mathcal{A}_i$.min
    assign u^* to partition i: $U_i \leftarrow U_i \cup \{u^*\}$
    remove u from U: $U \leftarrow U \setminus \{u^*\}$
    for $j = 1, \ldots, m$ **do**
        remove u^* from $\mathcal{A}_j$ by updating the links
    end for
    for $v \in \mathcal{N}(u^*) \setminus S_i$ **do**
        $S_i \leftarrow S_i \cup \{v\}$
        for $u \in \mathcal{N}(v) \cap U$ **do**
            $\mathcal{A}_i(u) \leftarrow \mathcal{A}_i(u) - 1$
            update the links in $\mathcal{A}_i$
        end for
    end for
end while

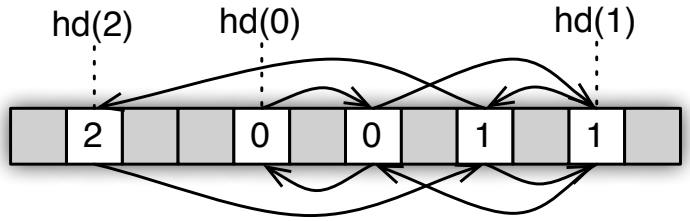


Figure 9.4: The data structure to store the vertex costs. It is an array with the i-th entry for vertex u_i, where assigned vertices are marked with gray color. The pointers and the doubly-linked list provide faster access to the data.

$S_i \subseteq V$, which are the neighbor sets assigned to i. The outputs are the m subsets that form the partition $U = \bigcup_i U_i$ and the updated subsets S_i.

Storing the vertex costs Note that if we subtract the constant $|\mathcal{N}(U_i)| + \alpha(|U_i| + 1)$ from $g_i(u)$,

we obtain the cost associated with vertex u as

$$\text{cost}_i(u) := |\mathcal{N}(U_i \cup \{u\})| - |\mathcal{N}(U_i)|, \tag{9.9}$$

which reflects the number of new vertices that would be added to the neighbor set of partition U_i because of adding vertex u to U_i.

When adding a vertex u to the subset U_i, this cost is changed only for a few other vertices. In other words, let $\Delta_i := \{v \in \mathcal{N}(u) : v \notin \mathcal{N}(U_i)\}$ denote the set of new vertices that are added into the neighbor set of U_i, the only vertices that will have their costs reduced are those connected to vertices in Δ_i. Due to the sparsity of the graph, this is often a small subset. Therefore, the computation overhead of storing and updating the vertex costs could be much smaller than re-computing all the costs at each time.

Fast-to-locate data structure Figure 9.4 illustrates the data structure for storing the vertex costs. For each subset U_i, we use an array $\mathcal{A}_i$ to store the vertex costs, and the j-th entry is for vertex j, namely $\mathcal{A}_i(u_j) = \text{cost}_i(u_j)$. On top of the array, we impose a doubly-linked list sorted according to increasing cost so that we can locate the lowest-cost vertex quickly.

When the cost of a vertex is modified), we update the doubly-linked list to restore the ordering. Note that the vertex cost is no larger than its degree, and most large scale graphs have a heavy-tailed degree distribution. Therefore the costs of a large portion of vertices are small integers. We store a small array of "head" pointers to the locations in the list where the cost jumps to $0, 1, 2, ..., \theta$. These pointers can speed up locating of elements in the list when updating the costs. In practice, we found that setting $\theta = 1000$ is sufficient to cover more than 99% of vertex costs.

The initial $\mathcal{A}_i(u)$'s can be computed in $\mathcal{O}(|E|)$ time and then be ordered in $\mathcal{O}(|U|)$ by counting sort, as they are integers upper bounded by the maximal vertex degree. The most expensive part of Algorithm 13 is updating $\mathcal{A}_i$ in line 22, which are evaluated at most $k|E|$ times as a vertex $v \in V$ together with its neighbors are accessed at most once. The time complexity of updating the doubly-linked lists is $\mathcal{O}(1)$. The cost to access the j-th vertex is $\mathcal{O}(1)$ due to the sequential storing on an array. Finding a vertex with the minimal value or removing a vertex from the list is also in $\mathcal{O}(1)$ time because of the doubly links. Keeping the list ordered after decreasing a vertex cost by 1 is $\mathcal{O}(1)$ in most cases ($\mathcal{O}(|U|)$ for the worst case), as discussed above, by using the cached head pointers. Therefore, the average time complexity of Algorithm 13 is $\mathcal{O}(m|E|)$.

9.4.2 Divide into Subgraphs

In line 10 of Algorithm 10 we sample a subset R in order to keep the partitions balanced. The constraint $|S| = 1$ introduced in Section 9.4.1 ensures that only a single vertex is assigned each time and thus we are guaranteed with the balancedness of the partition. In this case, we would like to sample as many vertices as possible to enlarge the search range of the optimal u^*. However, sampling is still appealing since a proper sample size can significantly reduce the computational time, while only slightly affecting the partition quality.

To address this issue, we first randomly divide U into b blocks, and then construct the b subgraphs by including the neighbor vertices from V and the corresponding edges. Let $\{G_j\}_{j=1}^b$ denote the b subgraphs, and let $\{S_i\}_{i=1}^m$ denote the initialized neighbor sets. We then apply

Algorithm 13 to partition these subgraphs sequentially. Specifically, the inputs are G_j and S_i and the outputs are the partition $\bigcup_{i=1}^{m} U_{i,j}$ for G_j and the updated set S_i. Finally we take a union of the solutions for all the subgraphs to obtain the final partition of U as $U_i = \bigcup_{j=1}^{b} U_{i,j}$.

Note that instead of sampling a new subgraph R for each single vertex assignment as in line 10 of Algorithm 10, our implementation actually fixes the subgraphs from the beginning. This strategy has several advantages. First, it exploits the efficient implementation of Algorithm 13 to partition each subgraph. Next, it is convenient to parallelize the algorithm with the subgraphs. Finally, this strategy is I/O efficient, as we only keep the current subgraph in memory, which makes it possible to partition graphs of sizes much larger than the physical memory.

Also note that the number of subgraphs b trades off partition quality and computational efficiency. In the case where $b = 1$, the vertex assigned to a partition is the best one from all unassigned vertices. It is, however, the most time consuming. On the other extreme, by setting $b = |U|$ we recover the solution of random partition with time complexity reduced to $\mathcal{O}(|E|)$. A moderate b serves to balance the computation time and partition quality.

9.4.3 Parallelization with Parameter Server

For very large graphs, parallelization of the algorithm is desirable. Parsa uses Parameter Server (see Chapter 3) to parallelize the partition process by having multiple nodes work on different subgraphs in parallel using the shared neighbor sets. More specifically, we define three kinds of nodes:

The scheduler, which issues partitioning tasks to workers and monitors their progress.

Server nodes, which maintain the global shared neighbor sets and process the push / pull requests from the worker nodes.

Worker nodes, which solve subgraphs partitioning in parallel. A worker node reads a subgraph from the (distributed) file system, and pulls the newest neighbor sets associated with this subgraph from the server nodes. It applies Algorithm 13 to partition this subgraph and pushes the modified neighbor sets to the servers in the end.

There are several details worth noticing: First, the data communication in Parameter Server is asynchronous, and we impose a maximal allowed delay τ to control the data consistency. Second, the worker node may only push the changes of the neighbor sets to the servers in order to save the communication traffic. Finally, a worker node can start a separate data pre-fetching thread to run step 3, 4 and 5 in Algorithm 12 in order to improve computation efficiency.

9.4.4 Initialize the Neighbor Sets

Note that the neighbor sets play a similar role as cluster centers on clustering methods, and a good initialization of the neighbor sets leads to better partition results. One typical choice of initialization is the empty set, which prefers to assign the vertices with small degrees first. Partitioning these vertices, however, often helps little, or even degrades, the following assignment. Next, we discuss a few heuristics for initializing the neighbor sets.

Individual initialization Given a graph that has been divided into b subgraphs. For a fixed constant a, we sequentially partition the first a subgraphs, and each time we only use the

partition of the preceding subgraph to initialize the neighbor set for the current subgraph. Then, we forsake the partitioning results of the first $(a-1)$ subgraphs. Starting with the partition of the a-th subgraph, we sequentially partition the other $(b-1)$ subgraphs, and each time we use the partition of all the processed subgraphs to initialize the neighbor set for the current subgraph.

Global initialization In parallel partitioning, we first sample a small subgraph and solve the partition problem for this subgraph. The neighbor sets obtained with this subgraph partition is then used as an initialization by all other worker nodes.

Incremental partitioning When data comes in an incremental or streaming way, we can use the partition results from outdated data as an initialization of the neighbor sets.

9.5 Experiments

9.5.1 Setup

We implement Parsa with our Parameter Server, and the source codes are available at `https://github.com/mli/parsa`. We compared Parsa with the popular vertex-cut graph partition toolboxes Zoltan[3] and PaToH[4]. We also benchmark performance with the well-known graph partition package METIS[5] and the greedy algorithm adopted by Powergraph[6]. All of these algorithms are implemented in C/C++.

We compare the runtime and partition results of these algorithms. The default measurement of the partition quality is the maximal individual traffic volume. We measure the improvement over random partition by

$$\frac{\text{random-partition} - \text{proposed-partition}}{\text{proposed-partition}} \times 100\%. \tag{9.10}$$

Thus a 100% improvement means that the traffic volume or memory footprint is 50% of that achieved by random partitioning.

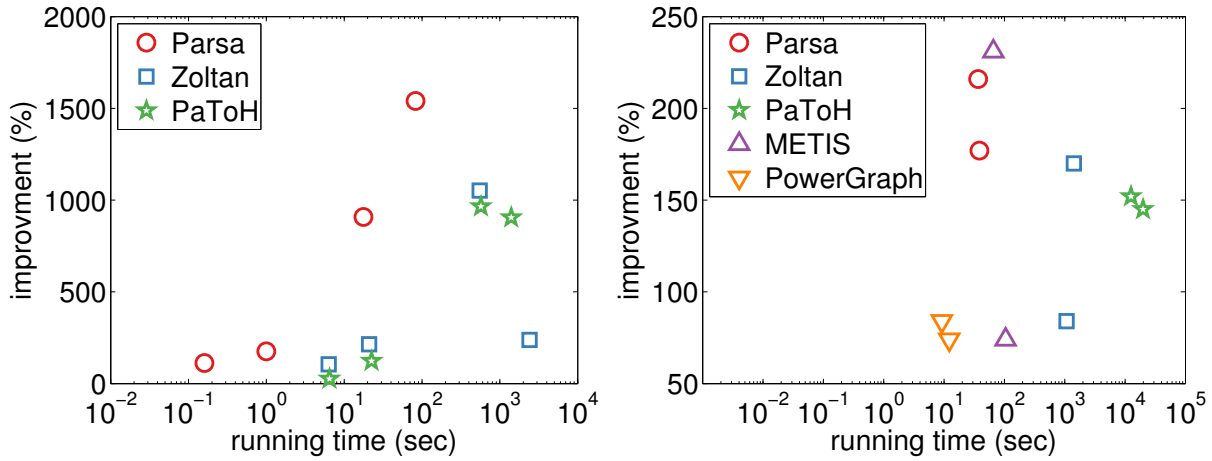
The default number of partitions was set to 16. As Parsa is a randomized algorithm, we average the results over 10 trials. For single thread experiments we use the desktop Desktop with an Intel i7 3.4GHz CPU, while for the parallel experiments we use the university cluster CampusA with 16 machines, each with an Intel Xeon 2.4GHz CPU and 1 Gigabit Ethernet.

9.5.2 Performance of Parsa

[3] `http://www.cs.sandia.gov/Zoltan/`
[4] `http://bmi.osu.edu/~umit/software.html`
[5] `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`
[6] `http://graphlab.org/downloads/`

datasets	PowerGraph				METIS				PaToH				Zoltan				Parsa			
	imprv (%)			time	imprv (%)			time	imprv (%)			time	imprv (%)			time	imprv (%)			time
	$M_{\max}$	$T_{\max}$	T_{sum}	(sec)	$M_{\max}$	$T_{\max}$	T_{sum}	(sec)	$M_{\max}$	$T_{\max}$	T_{sum}	(sec)	$M_{\max}$	$T_{\max}$	T_{sum}	(sec)	$M_{\max}$	$T_{\max}$	T_{sum}	(sec)
RCV1	-	-	-	-	-	-	-	-	19	26	279	7	17	105	154	6	33	112	108	0.2
News20	-	-	-	-	-	-	-	-	2	123	389	23	0	214	267	21	23	187	155	1
CTRa	-	-	-	-	-	-	-	-	8	446	970	571	70	1052	1211	551	91	922	913	18
KDD14	-	-	-	-	-	-	-	-	54	905	1102	1401	4	238	313	2409	120	1973	1978	89
LiveJournal	61	84	89	9	185	231	279	65	103	152	160	3.5h	50	84	386	1072	142	216	214	37
Orkut	55	74	78	12	56	74	103	104	87	145	150	5.5h	49	170	180	1413	105	177	121	39

Table 9.1: Improvements (%) compared to random partition on the maximal individual memory footprint $M_{\max}$, maximal individual traffic volumes $T_{\max}$, and total traffic volumes T_{sum} together with running times (in sec) on 16-partition. The best results are colored by Red and the second best by Green. Only 1% of CTRa is used.



(a) Text datasets (b) Social Networks

Figure 9.5: Visualization of Table 9.1.

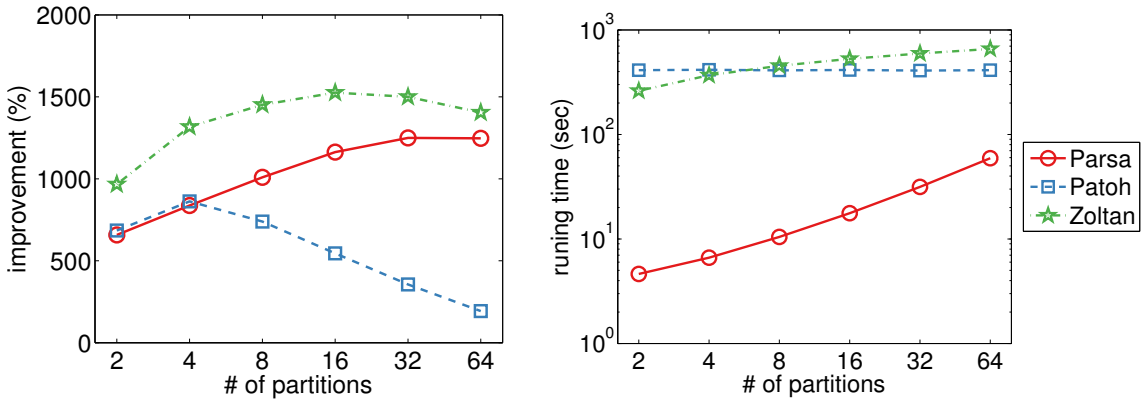
In Table 9.1, we compare the performance of Parsa with other algorithms on different datasets. We record the CPU time for running each algorithm. The time for loading the data is not included, as it varies for different data formats specific to the algorithms. The improvements are measured according to (9.10) for maximal individual memory footprint and traffic volume, together with the total traffic volume, which is the objective for both PaToH and Zoltan. Since neither METIS nor PowerGraph handles general sparse matrices, we only report the results on the dataset of social networks. The number of partitions is 16, and the hyper parameters of Parsa are set to $a = b = 16$. Recall that b is the number of subgraphs and a is the number of subgraphs used for individual initialization. In Figure 9.5, we plot both improvements on the maximal individual traffic volume and the runtime.

Observe that Parsa is not only 20x faster than PaToH and Zoltan, but also produces more stable partition results, with reduced memory footprint. METIS outperforms Parsa on one out of the two social networks but consumes twice of the CPU time. PowerGraph is the fastest but has poor partition quality. Parsa also outperforms other algorithms in terms of both maximal individual traffic volume and total traffic volume.

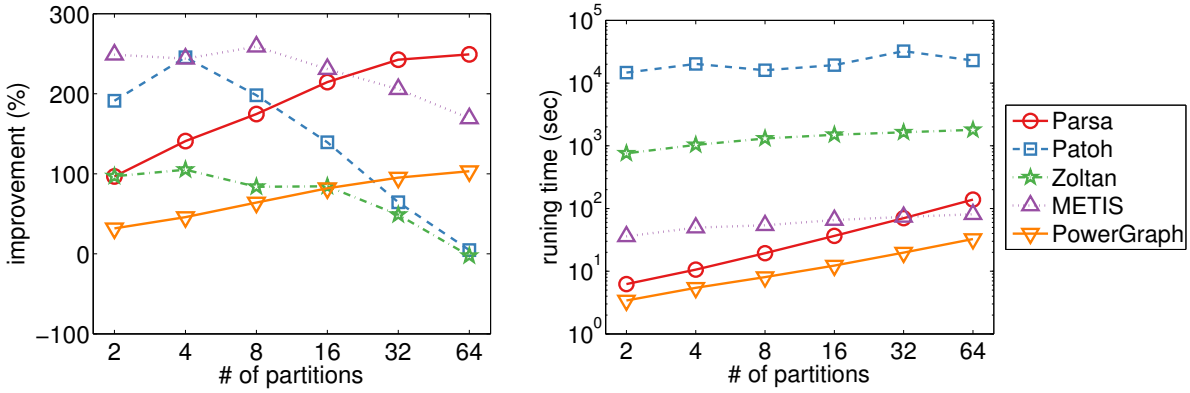
As shown in Figure 9.6, when the number of subsets in the partition increases, the runtime of those recursive-bisection-based algorithms (METIS, PaToH, and Zoltan) does not change much, but their partition quality degrades. In contrast, for Parsa and Powergraph, their run time increases linearly with m, but their partition quality actually improves.

Next, we tune the hyper parameters of Parsa and observe how its performance changes.

Vary the number of subgraphs and the percentage of data used for initialization We examine the effect of number of subgraphs (Section 9.4.2) and the neighbor set initialization (Section 9.4.4). We first consider the single thread case, with the empty neighbor set initialization. We vary the number of subgraphs b and also the number of subgraphs a used for individual initialization. Figure 9.7 shows the results on representative text dataset CTRa and social network dataset LiveJournal, where the x-axis is $a/b \times 100\%$, namely the percent of data used for con-



(a) Dataset: 1% of CTRa



(b) Dataset: LiveJournal

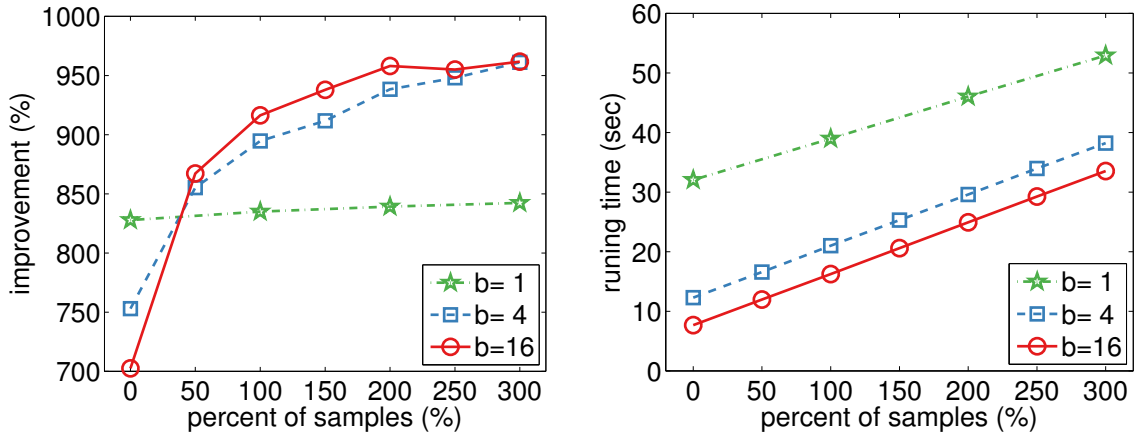
Figure 9.6: Partition quality and runtime for different number of partitions.

structing the initialization. We can observe that using more data for initialization improves the partition quality. In particular, for $b > 1$, there is a stable 20% improvement when at least 100% data are used.

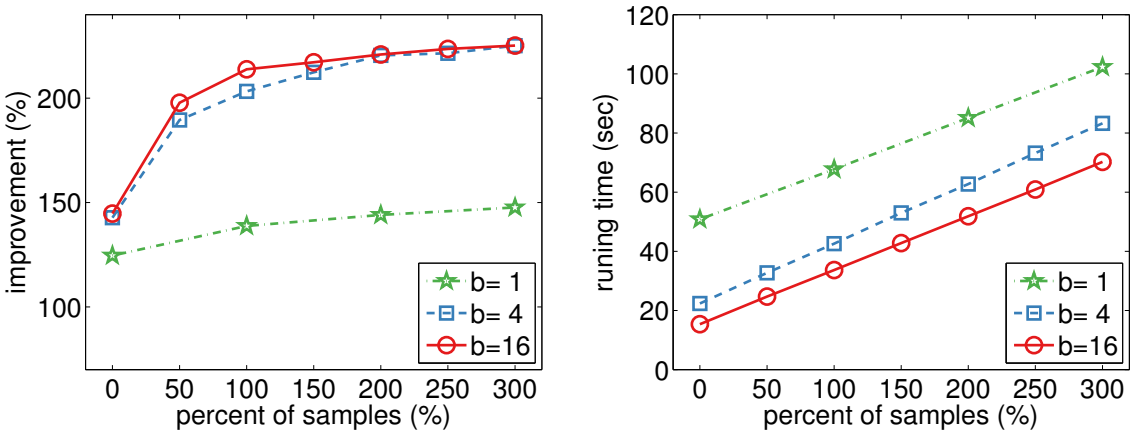
With empty neighbor set initialization, using small subgraphs has a positive effect on the partition results for LiveJournal, but not for CTRa. The reason is that Parsa tends to assign vertices with small degrees first when starting with empty neighbor sets. Those vertices offer little or no benefit for the subsequent assignment. LiveJournal has many more sparse vertices than CTRa due to the power law distribution of the degrees, and partitioning small subgraphs helps to prevent the sparse vertices coming into the partition too early.

Our nontrivial individual initialization addresses this issue by only using those partition results for setting the initial neighbor sets. Notably, with $a > \frac{1}{2}b$ in Figure 9.7, small subgraphs improve the partition results on both CTRa and LiveJournal. This occurs because, when using the same percentage of data for initialization, the neighbor sets are reset more often with a small b.

Figure 9.7 also shows that the runtime actually decreases with b. This is because splitting into more blocks (larger b) narrows the search range for adding vertices and thus reduces the cost of operating the doubly-linked list. On the other hand, the runtime increases linearly as we use



(a) Dataset: 1% of CTRa



(b) Dataset: LiveJournal

Figure 9.7: Partition quality and runtime for different percentage of data used in initialization (Single thread implementation).

more subgraphs for individual initialization, but the partition quality benefits of doing so appear worthwhile up to performing two passes (100% samples).

We also examine the performance of parallel implementation with non-empty initial neighbor sets. We use 4 workers to partition a subset of CTRa containing 1 billion of edges and use one worker to partition an even smaller subgraph with size n^o to obtain the starting neighbor sets. Figure 9.8 shows that the partition quality is significantly improved even when only 0.1% of the data are used for the initialization. In addition, although initialization takes extra time, the total runtime is minimized at $n^o > 0$. This is because a good initialization of the neighbor sets reduces the cost of operating the doubly linked lists, thereby reduces the overall runtime.

Scalability We test the scalability of Parsa on full CTRa with 10 billions of edges. We run 4 workers and 4 servers at each machine. Figure 9.9 shows that the speedup scales almost linearly with the number of machines and is close to the ideal case. In particular, we obtained a 13.7x

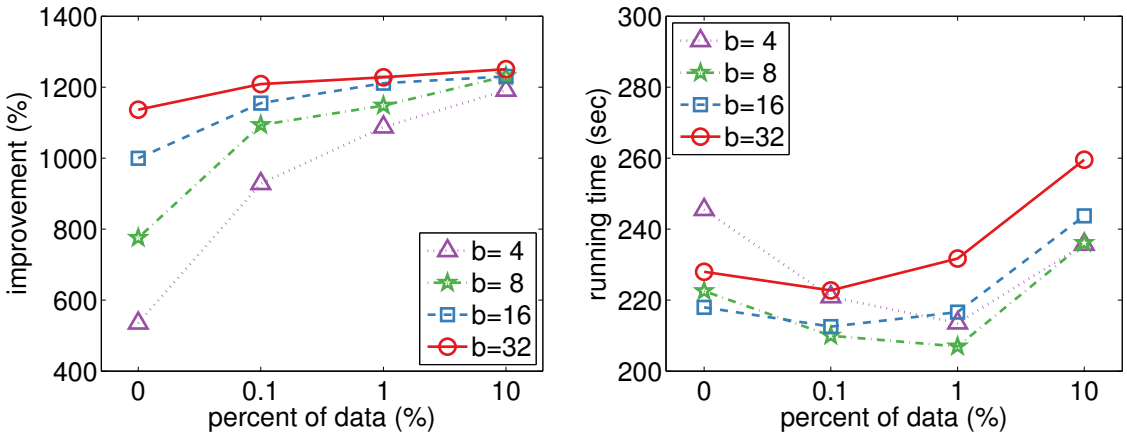


Figure 9.8: Partition quality and runtime for different percentage of data used in initialization (parallel implementation).

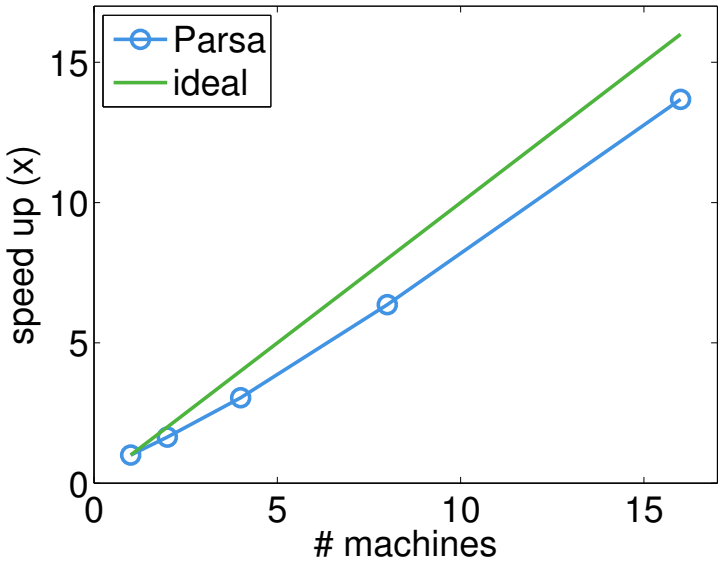


Figure 9.9: Speedup of Parsa when the number of machines increases (on dataset CTRa).

speedup by increasing the number of machines from 1 to 16. The main reason that Parsa scales well is the eventual consistency model ($\tau = \infty$), where each worker does not need to wait till the results from previous iterations are pushed successfully. Therefore, workers fully utilize the computing resource and network bandwidth, without the overhead of data synchronization.

This consistency model potentially leads to inconsistency of the neighbor sets between workers, which Parsa is quite robust to. In our experiment, when the number of machines increases from 1 to 16 (number of workers increases from 4 to 64), the quality of the partition result decreases by at most 5%. We believe that the reason is twofold. First, the initial neighbor sets obtained with a small subgraph give all workers a consistent initialization, which may already contain most of the hubs (large degree vertices) in V. Second, after a worker partitions a sub-

method	partition	inference	total
random	0	1.43	1.43
Parsa	0.07	0.84	0.91

Table 9.2: Time in hours for solving ℓ_1-regularized logistic regression. We runs 45 data passes using 16 machines for the dataset CTRa.

graph, the updates to the neighbor sets mainly include vertices with small degree. Thanks to the sparsity of the low degree vertices, the conflicts among workers is small and thus the inconsistency does not drastically affect the final result.

Accelerate Distributed Machine Learning Finally we examine how much Parsa can accelerate distributed machine learning applications with the better data and parameter placement. We consider ℓ_1-regularized logistic regression with the distributed inference algorithm DBPG (see Chapter 6 for more details).

We run DBPG on CTRa using 16 machines as the baseline, with all optimization options enabled. Then we apply Parsa to partition CTRa into 16 parts and run the algorithm again. Table 9.2 compares the performance of the above two methods. By random partitioning, DBPG stops after passing the data 45 times and uses 1.43 hours. On the other hand, Parsa uses 4 minutes to partition the data and then accelerates DBPG to 0.84 hour, which is an overall 1.6x speedup.

By random partition, only 6% network traffic between servers and workers happens locally. Although Chapter 6 reports nearly zero communication cost on DBPG, we observe that a significantly part of time is spent on data synchronization. The reason is twofold. First, in Chapter 6 we use data pre-processing for CTRb to remove tail features (tail vertices in V). That is, the feature to examples ratio is 2.83 for CTRa while 0.38 for CTRb. However, here we feed the raw data into the algorithm and let the ℓ_1-regularizer work out feature selection automatically, which often leads to better machine learning model but induces more network traffic. Second, the network bandwidth of the university cluster in our experiment is 20 times less than the industrial datacenter used in Chapter 6, and thus the communication cost can not be ignored here. We observe that with the problem partitioning, the total data communication is reduced from 4.5TB to 4TB, and the ratio of inner-machine traffic increases from 6% to 92%. Over all, the inter-machine communications decreases by more than 90%.

Chapter 10

DiFacto: Scaling Distributed Factorization Machines

10.1 Introduction

Nonlinear models for recommendation and estimation have spurred significant interest in recent years. The growing body in deep learning [42], kernel methods [81], and decision trees [148] bears witness of this development. For recommender systems and polynomial generalized linear models Factorization Machines (FM), [114, 115] offer a computationally efficient and powerful alternative. They achieve excellent results by using a low-rank expansion of the higher degree polynomial terms. Moreover, they offer a principled framework for a number of feature space heuristics in recommender systems, such as bias, features, cold-start strategies, and temporal models. This framework makes them a very attractive target for statistical modeling on high-dimensional sparse data occurring in many settings such as computational advertising, personalization, user profiling, recommendation, and search.

Unfortunately, despite the low-rank expansions, the memory cost remains tremendous for real-world settings. This occurs because each feature, each user, and each object need to be embedded into a low-dimensional space. A quick calculation even on modest datasets such as Criteo shows that we have up to 1 billion features. Realistic problems, such as CTRb, can have up to 10^{11} terms. Even a modest 100 dimensional representation would require data (parameters, preconditioners, auxiliary key storage) in the order of 1TB. This is computationally intractable on a single machine, especially when tackling industrial scale problems. Moreover, even multiple machines are heavily taxed by the large memory footprint. In order to address the above issues, we need:

1. A compact model with low computation and communication cost which still provides high-quality embeddings;
2. An efficient mechanism of problem partitioning, optimization, and high performance communications protocols, targeting for distributed optimization.

In this paper, we first make the key observation that the importance of features in real datasets is not uniform, which deserves adaptive model capacities. We propose a refined FM model, called DiFacto (*Di*stributed *Facto*rization). DiFacto is based on the Parameter Server framework (for

details please see Chapter 3). The algorithm adaptively chooses the effective embedding dimension and regularization for each feature, based on the feature frequency count and sparse regularization. The resulted model is up to 100x more compact than conventional FM and even provides better generalization accuracy. We prove that the proposed algorithm converges for this highly challenging nonconvex objective function even under asynchronous updates. Experiments show that for sparse data, DiFacto can scale up to handle billions of examples and features. To the best of our knowledge, our results describe the largest statistical model ever computed as in [7] by at least one order of magnitude. Moreover, our algorithms require only modest resources on a per-machine basis.

10.2 Background

10.2.1 Objectives

In this chapter we discuss two related goals — recommendation and prediction. The distinction between those two problems is partly due to different notational conventions, and partly due to slightly different objectives.

In recommender systems [15] one is typically given *pairs* of entities, say user u and movie m for which a rating $y(u, m)$ needs to be estimated. The quality of an estimate of the rating, denoted by $g(u, m)$, is evaluated, e.g. by the squared discrepancy between rating and estimate, i.e. $\frac{1}{2}(y(u, m) - g(u, m))^2$, or by the quality of the relative ordering of ratings, or by per-session ordering of preferences. For an exhaustive summary of such objectives see e.g. [101]. In short, we are interested in the performance of $g(u, m)$ relative to $y(u, m)$. Sometimes when we have additional feature z_u, z_m for u and m and context c, we include them in the estimate and denote the estimate by $g(u, z_u, m, z_m, c)$. In the following we use the shorthand

$$x := (u, z_u, m, z_m, c) \quad \text{and} \quad g(x) := g(u, z_u, m, z_m, c) \tag{10.1}$$

to indicate the *function* that takes the features in x as inputs and outputs the estimate $g(x)$.

In prediction we are interested in the slightly more mundane goal of obtaining $g(x)$, given pairs of *feature* x and *label* y. Typical goals are to minimize the squared deviation $\frac{1}{2}(y - g(x))^2$ or the log-likelihood of a particular label $\log(1 + e^{-yg(x)})$. Note that y and $g(x)$ need not be of the same data type, and this was exploited substantially in structured estimation [136].

In the following we will not make major distinctions between the two problems of recommendation and prediction. Instead, we simply assume that x is either a set of covariates, such as the words occurring in a document for which the click through rate (CTR) of an advertisement is sought, or a set of recommendation and rating parameters. The quality will be evaluated via a loss function $\ell(x, y, g(x))$, yielding the risk functional

$$\frac{1}{n} \sum_{i=1}^{n} \ell(x_i, y_i, g(x_i)). \tag{10.2}$$

10.2.2 Factorization Machine

The typical setting in our context is that the covariates $x \in \mathbb{R}^p$ lives in a very high p-dimensional space and is sparse. For instance, in recommender systems x might only contain two nonzero terms — an indicator for the user, and another indicator for the movie, i.e. $x_u = 1$ and $x_m = 1$. The consequence of such a setting is that linear models are of limited use. For instance, in a recommender system this would amount to

$$g(x) = \langle w, x \rangle = w_u + w_m. \tag{10.3}$$

In other words, this is the trivial case where recommendations are simply the sum of user and movie biases. Nonetheless, due to the very high dimensionality of the data and the associated uncertainty in determining even just linear parameters, in CTR estimation problems [99], such models are quite popular. In particular, even a quadratic model, which require $O(p^2)$ parameters, is too expensive to estimate, since typically the number of observations $n \ll p^2$.

Rendle and the coauthors [114, 115] introduced a principled strategy for alleviating this problem via a low-rank expansion instead of a general high-dimensional expansion. That is, they propose

$$g(x) = \langle w, x \rangle + \sum_{i<j} x_i x_j \operatorname{tr}\left(V_i^{(2)} \otimes V_j^{(2)}\right) + \sum_{i<j<k} x_i x_j x_k \operatorname{tr}\left(V_i^{(3)} \otimes V_j^{(3)} \otimes V_k\right) + \dots, \tag{10.4}$$

where $V^{(i)} \in \mathbb{R}^{p \times k_i}$ are the embedding matrices. In other words, Factorization Machines (FM) uses a low-dimensional embedding of the (typically very sparse set of) features in x to a much smaller k_i-dimensional space via the embedding matrices. For the purpose of the current paper we limit ourselves to expansions up to second order. This simplifies the exposition. Also, it is sufficient given the sparsity of the problem as we will discuss subsequently. Hence we may rewrite (10.4) as

$$g(x) = \langle w, x \rangle + \frac{1}{2} \|Vx\|_2^2 - \sum_{i=1}^{d} x_i^2 \|V_i\|_2^2, \tag{10.5}$$

where we used the shorthand $V := V^{(2)}$ and $k_2 = k$, and V_i is the i-th column in V. Here we used the polarization equality to rewrite the ordered sum, and we used the fact that $\operatorname{tr}(a \otimes b) = \langle a, b \rangle$. Note that (10.5) can be computed efficiently in $O(pk)$ time, requiring $O(pk)$ storage. This also illustrates a shortcoming of Factorization Machines.

Lemma 21 *Assume that we estimate (10.5) with an embedding dimension k. Then if for feature i we only observe $x_i \neq 0$ for less than $k+1$ times, the problem of estimating (w_i, V_i) is underdetermined.*

Proof. This follows directly from linear algebra: whenever $x_i = 0$ the terms v_i, V_i do not occur in (10.5). To solve a linear system of $k+1$ variables we need at least the same number of constraints. ■

However, we need to use $O(k)$ memory even for variables that hardly ever occur. Given the power-law nature of many natural feature distributions, this is clearly undesirable. Large numbers of variables are not just hard to estimate and hard to store, they are also cumbersome in terms of optimization purposes, since they slow down the convergence and need to be addressed via additional stabilizers (regularization). Moreover, usually for large scale problems, the number of parameters significantly exceeds the amount of available memory on a computer, which calls for distributed *representations* and *optimization algorithms*.

10.3 Statistical Model

DiFactor considers the following optimization problem:

$$\min_{w,V} \frac{1}{|n|} \sum_{i=1}^{n} \ell(g(x_i), y_i) + \lambda_1 \|w\|_1 + \frac{1}{2} \sum_{i=1}^{p} \left[\lambda_i w_i^2 + \mu_i \|V_i\|_2^2\right], \tag{10.6}$$

$$\text{subject to } V_{ij} = 0 \text{ for } j > k_i, \tag{10.7}$$

where the constants λ_1, λ_i, μ_i and k_i are coefficients for regularization. This model differs in two key parts from standard FM models. First, we add frequency adaptive regularization to better model the possible nonuniform sparsity pattern in the data. Second, we use sparsity regularization and memory adaptive constraints to control the model size, which benefits both the statistical model and system performance.

In the rest of this section, we will discuss the details of the above formulation and introduce the model capacity control strategies adopted by DiFacto.

10.3.1 Memory Adaptive Constraints

The first step in making factorization machines tractable is to modify the expansion in (10.4) in line with Lemma 21. Whenever we have insufficient evidence of a feature occurring, it makes no sense to allocate much capacity to it. More to the point, instead of allocating a *fixed* number of dimensions k to *each* feature i regardless of how frequently x_i is not equal to 0, we make this number dependent on the number of times $x_i \neq 0$ on the training set. Define n_i to be:

$$n_i := |\{x : x_i \neq 0\}|. \tag{10.8}$$

Given a set of dimensionalities $\{k_i\}$ we impose the constraint $V_{ij} = 0$ for all $j > k_i$. This forces the infrequently occurring features to assume a rather lower-dimensional embedding using only k_i dimensions rather than the full set $k > k_i$. A simple choice for the dimensions is

$$k_i = \begin{cases} k & \text{if } n_i > r \\ 0 & \text{otherwise} \end{cases}, \tag{10.9}$$

where we can set $k^{(1)} = r^{(1)} = 10^3$ and $k^{(2)} = r^{(2)} = 100$, and this differentiated setting yields a larger embedding for the more frequent keys.

10.3.2 Sparse Regularization

Note that the above strategy is entirely independent of the actual problem we solve. We also need a data adaptive capacity control mechanism that depends on the predictive power of features relative to the labels y_i. A popular choice is ℓ_1 regularization [139], which has been widely used in linear models for high dimensional data such as computational advertising [99]. In ℓ_1 regularization, we add a penalty function to the optimization objective:

$$h(w) = \lambda_1 \|w\|_1 ,$$

where λ_1 controls the degree of sparsity. The sparse model induced by the ℓ_1 regularization not only penalizes complex models, but also reduces the computation cost of the gradients and saves the communication traffic. It results in a smaller final model which further makes it easier to deploy for online service. We can apply a similar structured sparsity [104] on V to encourage sparse solutions:

$$h(w, V) = \sum_i \left[w_i^2 + \|V_i\|_2^2\right]^{\frac{1}{2}} + \|V_i\|_2 . \tag{10.10}$$

In this case the derivative of the penalty vanishes at $w_i = 0$, provided that $V_i \neq 0$. Unfortunately, this penalty is harder to handle from the system perspective. Hence we replace it by a rather straightforward approximation that *directly* implements (10.10) by enforcing $V_i = 0$ whenever $w_i = 0$. Our experiments show that this strategy is by no means detrimental to the overall outcome of the estimation, and actually it also closely resembles the ANOVA decomposition hierarchy proposed by [145].

10.3.3 Frequency Adaptive Regularization

It is well known that adaptively penalizing terms occurring at different frequency can lead to improved generalization performance [128]. For instance, in collaborative filtering, the strategy to penalize frequent terms *more aggressively*, e.g. by performing shrinkage only whenever a user (or a movie) is being updated, has proven successful. In DiFacto, we use a slightly more general and more flexible method for capacity control: we shrink whenever an feature occurs in a minibatch. The consequence is that parameters associated with more frequent features are less over-regularized. More specifically, we can define the penalty function with feature specific coefficients as:

$$h(w, V) = \frac{1}{2} \sum_i \left[\lambda_i w_i^2 + \mu_i \|V_i\|_2^2\right] . \tag{10.11}$$

Lemma 22 (Dynamic Regularization) *Assume that we solve a factorization machines problem*

with the following updates in a minibatch update setting: for each feature i,

$$I_i \leftarrow I\left\{[x_j]_i \neq 0 \text{ for some } (x_j, y_j) \in B\right\}, \tag{10.12}$$

$$w_i \leftarrow w_i - \frac{\eta_t}{b} \sum_{(x_j,y_j)\in B} \partial_{w_i} \ell(g(x_j), y_j) - \eta_t \lambda w_i I_i, \tag{10.13}$$

$$V_i \leftarrow V_i - \frac{\eta_t}{b} \sum_{(x_j,y_j)\in B} \partial_{V_i} \ell(g(x_j), y_j) - \eta_t \mu V_i I_i, \tag{10.14}$$

where B denotes the minibatch of data and b is the size of the minibatch, and I_i is the indicator for whether or not feature i appears in this minibatch. Then effective coefficients for regularization function is given by:

$$\lambda_i = \lambda \rho_i \quad \textit{and} \quad \mu_i = \mu \rho_i,$$

where $\rho_i = 1 - (1 - n_i/n)^b \approx n_i b/n$.

Proof. The probability that a particular feature occurs in a random minibatch is $\rho_i = 1 - (1 - n_i/n)^b$. Observe that while the amount of regularization on the minibatches is not independent, it is additive (and exchangeable). Hence the expected amount of regularizations are $\rho_i \lambda$ and $\rho_i \mu$, respectively. Expanding the Taylor series in ρ_i yields the approximation. ■

Note that for $b = 1$ we obtain the conventional frequency dependent regularization, whereas for $b = n$ we obtain the Frobenius regularization. Choosing the minibatch size conveniently allows us to interpolate between the two extreme cases.

10.4 Distributed Optimization

The optimization problem in (10.6)o is challenging, especially when large models bring large communication cost for distributed computing. We focus on asynchronous optimization method, which hides synchronization cost by communicating and computing in parallel.

10.4.1 Asynchronous Stochastic Gradient Descent

To solve the optimization problem in (10.6), we use asynchronous stochastic gradient descent (SGD). In a distributed computing environment, a worker can calculate the gradients of the loss function. First, compute the partial gradient of the estimator $g(\cdot)$ defined in (10.5):

$$\partial_{w_i} g(x, w, V) = x_i \tag{10.15}$$

$$\partial_{V_{ij}} g(x, w, V) = x_i \left[Vx\right]_j - x_i^2 V_{ij}. \tag{10.16}$$

Note that the term Vx can be pre-computed. Invoke the chain rule, and we can compute the partial gradients of the loss function $\ell(\cdot)$. Let w_t and V_t denote the model stored at the server at time t. We denote the partial gradients of the loss function that the server received from a worker node at time t by:

$$g_t^\theta \leftarrow \partial_\theta \ell(g(x, w_{t-\tau}, V_{t-\tau}), y), \tag{10.17}$$

where θ can be either w or V, and τ is the time delay, indicating that the gradients were computed by the worker using the possibly outdated model at time $(t-\tau)$.

As discussed in Section 10.3.3, we use frequency adaptive regularization for w and V and the sparsity-inducing ℓ_1 regularization for w. In addition, we use AdaGrad [49] to better model the possible nonuniform sparsity in the data. Upon receiving the gradients, the server updates V_{ij} by:

$$\begin{aligned} n_{ij} &\leftarrow n_{ij} + \left[g_t^V\right]_{ij}^2, \\ V_{ij}(t+1) &\leftarrow [V_t]_{ij} - \frac{\eta_V}{\beta_V + \sqrt{n_{ij}}} \left(\left[g_t^V\right]_{ij} + \mu \left[V_t\right]_{ij} \right), \end{aligned} \tag{10.18}$$

where η_V and β_V are scalar constants, and n_{ij} is initialized to 0 at time 0. The update rule for w is slightly different from that of V due to the nonsmooth ℓ_1 regularizer. We adopted FTRL [97], which solves a "smoothed" proximal operator based on AdaGrad. Let η_w and β_w denote the global learning rate, the server updates w_i by:

$$\begin{aligned} \sigma_i &\leftarrow \frac{1}{\eta_w} \left(\sqrt{n_i + [g_t^w]_i^2} - \sqrt{n_i} \right), \\ z_i &\leftarrow z_i - [g_t^w]_i + \sigma_i [w_t]_i, \\ n_i &\leftarrow n_i + [g_t^w]_i^2, \\ [w_{t+1}]_i &\leftarrow \begin{cases} 0 & \text{if } |z_i| \le \lambda_1 \\ \left(\frac{\beta_w + \sqrt{n_i}}{\eta_w} + \lambda_2 \right)^{-1} (z_i - \text{sgn}(z_i)\lambda_1) & \text{otherwise} \end{cases}, \end{aligned} \tag{10.19}$$

where both n_i and z_i are set to 0 at time 0.

10.4.2 Convergence Analysis

In Algorithm 14, we sketch our DiFacto for solving (10.6). The convergence analysis for this non-convex and non-smooth optimization problem is highly non-trivial, especially in the face of asynchronous updates. Next, we provide a preliminary analysis only for a special case with $\lambda_1 = 0$. For notation simplicity we assume a fixed learning rate. We can extend the results to handle the adaptive learning rate by assuming n_{ij} in (10.18) is bounded (see [84] for more details).

In particular, we first show that the simplified algorithm has a $O(1/\sqrt{t})$ ergodic convergence rate (see, e.g. [56]). Due to the non-convexity, our results do not imply convergence to a KKT point. However, it is stronger than typical non-convex analysis in the sense that there is an explicit convergence rate. By carefully incorporating the specific properties of our problem, we obtain a convergence bound in Theorem 26 that explicitly and intuitively reveals the dependency on the data sparsity and the size of the minibatch, as well as on the more typical maximum delay parameter for the asynchronous update.

First, the following lemma shows that Algorithm 14 can be reduced to asynchronous SGD.

Lemma 23 *Let η be the fixed learning rate for both parameter w and V, and in addition, $\lambda_1 = 0$, then the update equations* (10.18) *and* (10.19) *for solving the problem* (10.6) *becomes plain asynchronous SGD.*

Proof. First of all, we note that despite the memory adaptive constraints $V_{ij} = 0$ for some i, j, we are essentially solving an *unconstrained* smooth optimization problem in a pre-defined coordinate subspace, which admits a simple (non-projected) SGD algorithm.

It remains to show that the stochastic gradient is unbiased. Since the minibatch is picked randomly, the expectation of the loss-function half of the partial stochastic gradient with respect to V is just the gradient. Since λ_i is frequency adaptively chosen, it does not affect the expectation of the gradient at all. Now we turn to the partial gradient with respect to w. Under the assumption that $\lambda_1 = 0$ and without AdaGrad it follows that $n_i = 0$. Hence for every $t \in \{1, \dots T\}$ the update equation (10.15) becomes

$$[w_{t+1}]_i \leftarrow \frac{\eta_w}{\beta_w} [z_{t+1}]_i \text{ where } [z_{t+1}]_i \leftarrow [z_t]_i - [g_t^w]_i.$$

Substitute in $[z_1]_i = \frac{\beta_w}{\eta_w}[w_1]_i$, this recursion is essentially the SGD update equation

$$[w_{t+1}]_i \leftarrow [w_t]_i - \frac{\eta_w}{\beta_w}[g_t^w]_i.$$

with learning rate $\eta = \frac{\eta_w}{\beta_w}$. The approximation being unbiased is again trivial since the data points are picked uniformly at random and the regularization weight μ_i are frequency adaptive. ■

Our analysis builds upon a generic argument in [88], which we state below.

Theorem 24 ([88, Theorem 2]) *Assume that the stochastic gradient is unbiased and has variance bounded by σ^2 and that the gradient functional $\nabla f(\cdot)$ is L-Lipschitz. Moreover, assume τ_t, the delay at time t, is upper bounded by τ. Assume the global optimal solution x^* exists with objective value $f^* > -\infty$.*

If the stepsize satisfies that

$$L\eta_t + 2L^2\tau\eta_t \sum_{\kappa=1}^{\tau} \eta_{t+\kappa} \leq 1 \quad \textit{for all } t = 1, ...,$$

we have ergodic convergence rate for the following algorithm

1. *pick i randomly from data points $1, ..., n$.*
2. *update $w_{t+1} = w_t - \eta_t \nabla f_i(w_{t-\tau_t})$.*

In particular, the following bound holds

$$\sum_{t=1}^{T} \eta_t \mathbb{E}(\|\nabla f(w_t)\|^2) \leq 2(f(w_1) - f(w^*)) + \sum_{t=1}^{T} \left(\eta_t^2 L + 2L^2 \eta_t \sum_{j=t-\tau}^{t-1} \eta_t^2 \right) \sigma^2. \tag{10.20}$$

Divide both sides by $\sum_t \eta_t$ and we can state our convergence result as a corollary of the above theorem after verifying all the conditions. We can slightly simplify it with a fixed learning rate.

Corollary 25 ([88, Corollary 2]) *Under the same assumptions of Theorem 24, if we use a constant stepsize*

$$\eta := \sqrt{\frac{f(x_1) - f(x^*)}{L\tau\sigma^2}},$$

for every integer $T \geq \frac{4L(f(x_1)-f(x^*))}{\sigma}(\tau+1)^2$*, we have the output of the SGD obeying*

$$\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}(\|\nabla f(x_t)\|^2) \leq 4\sqrt{\frac{(f(w_1) - f(w^*))L}{T}}\sigma. \tag{10.21}$$

We apply the above results to our problem setup to prove the following theorem, where the constants explicitly depend on the average sparsity of the data and size of the minibatch we use.

Theorem 26 *Define* $\phi_i(w) = \ell(g(x_i), y_i)$ *and*

$$\Phi(w) = \frac{1}{n}\sum_{i=1}^{n}\phi_i(w) + \sum_{j=1}^{p}[\lambda_j w_j^2 + \mu_j\|V_j\|^2].$$

Let the delay of the asynchronous gradient $\tau_t \leq \tau$ *for any update* t *and the minibatch size be* b*. Assume that the data* X *obeys the following property:* $\|X\|_\infty \leq 1$ *and* $s_i = \|x_i\|_0 \leq s$*. Assume that we work within a sublevel set of the problem such that* $\|[w_{supp(x_i)}, V_{supp(x_i \otimes x_i)}]\| \leq B_i$*. Furthermore, assume that the regularization parameter is sufficiently small, so that the regularization terms do not weigh more than the data terms in the gradients.*

Then there is a universal constant C *and a data (sparsity) dependent constant* K *defined to be* $K = \frac{1}{nb}\max_i(s_i B_i)^2\sum_{i=1}^{n}(s_i B_i)$*, such that setting stepsize* $\eta = \sqrt{C\phi(w_1)/\tau K}$ *results in a stochastic sequence of parameters that satisfies*

$$\min_{t\in\{1,\ldots,T\}}\mathbb{E}\|\nabla\phi(w_t)\|^2 \leq \frac{1}{T}\sum_{t=1}^{T}\mathbb{E}\|\nabla\phi(w_t)\|^2 \leq \ 4\sqrt{\frac{\phi(w_1)\tau K}{T}}. \tag{10.22}$$

We first interpret the result before stating the proof. First, the expected magnitude of the gradient is a intuitive measure of the distance from stationarity condition, which is weaker than typical measures in convex optimization such as primal suboptimality, mean square distance from the optimal solution. For "nice" loss functions, it should be in the same ball park (see, e.g. discussions in [56, 88]). Second, when the data is sparse or the minibatch size is big, we can afford using larger learning rate and hence get faster convergence. Note that the results capture the average sparsity so the quantity remains small even when a few data points are dense. This is especially important in face of the practical power-law distributions of data.

Proof of Theorem 26. The proof is a specialization of Corollary 25 to our problem, which involves only calculating the gradient Lipschitz constant and the variance of the stochastic gradient and upper bounding them using the intuitive quantities of interests.

By the chain rule

$$\partial_w\phi_i(w) = \frac{\partial}{\partial g(x_i)}\ell(g(x_i), y_i)\frac{\partial}{\partial w}g(x_i).$$

Since ℓ is logistic loss, $|\frac{\partial}{\partial g(x_i)}\ell(g(x_i), y)| \leq 1$, it follows from (10.15) and (10.16) that

$$\begin{aligned}\|\partial_w \phi_i(w)\| &\leq 1 \cdot \sqrt{\sum_j (\partial_{w_j} g(x_i))^2 + \sum_{j,\ell} (\partial_{V_{j\ell}} g(x_i))^2} \\ &= \sqrt{\sum_j x_i^2 + \sum_{j,\ell} (x_j [Vx]_\ell - x_j^2 V_{j\ell})^2} \\ &\leq \sqrt{s_i}\|x_i\|_\infty + s_i \|x_i [Vx_i]^T\|_{2,\infty} \leq s_i B_i\end{aligned}$$

Since the objective is differentiable, the Lipschitz constant can be obtained by upper bounding the gradient

$$\begin{aligned}\|\partial_w \phi(w)\| &\leq \left\| \frac{1}{n} \sum_{i=1}^n \partial_w \phi_i(w) + 2\Lambda w \right\| \\ &\leq \frac{1}{n} \sum_{i=1}^n s_i B_i + \|\Lambda w\|_2 \leq \frac{2}{n} \sum_{i=1}^n s_i B_i\end{aligned}$$

where Λ is a big diagonal matrix that contains the frequency adaptive regularization weights λ_j and μ_j and the last inequality holds whne these weights are small.

The variance of the stochastic gradient is taken over an iid minibatch of size b, and it can be trivially bounded using the boundedness of the gradient

$$\sigma^2 \leq \frac{\max_i \|\partial_w \phi_i(w)\|^2}{b} \leq \frac{\max_i (s_i B_i + B)^2}{b} \leq \frac{4 \max_i (s_i B_i)^2}{b}.$$

The last inequality is again simplification by under the assumptions that the regularization terms are small. Finally, we note that $\phi(w) \geq 0$ for any w, so $\phi(w^*) \geq 0$. We arrive at (10.22) by substituting these bounds into (10.21) in Theorem 25. The proof is complete by noting that the minimum is always smaller than the mean in a sequence of numbers. ■

10.4.3 Implementation

Algorithm 14 sketches our algorithm DiFacto. It consists of three parts, where the scheduler node runs the control logic, the server nodes coordinate the workers and update the model, and the worker nodes compute the gradients with local data.

The Scheduler issues commands, such as *process examples I* or *save the model* to worker and server nodes. It also monitors the progress of workers. Once a straggler or a dead node is detected, the scheduler will re-issue the command to another available node. Also, the scheduler checks whether the stopping criteria has been reached.

Server nodes maintain and update the model w and V. Due to the adaptive memory constraints, the elements of V_i contain a lot of zeros, and thus to reduce memory footprint a server node only needs to allocate physical memory to nonzero element of V_i. It also applies when a

Algorithm 14 Implement DiFacto in the parameter server

Scheduler Node:

Assume the data is partitioned into s parts $\bigcup_{k=1}^{s} I_k = \{1, \ldots, n\}$
for $t = 1$ **to** T **do**
    $I = \{I_1, \ldots, I_s\}$ and $A = \emptyset$
    while $I \neq \emptyset$ **do**
        switch detected event from worker k **do**
            case idle
                Pick $I_i \in I \setminus A$ and assign I_i to worker k
                $A = A \cup \{I_i\}$,
            case finished I_i
                $I = I \setminus \{I_i\}$
            case dead or timeout
                $A = A \setminus \{I_i\}$,
    end while
end for

Worker k:

Receive command "processing I_i" from the scheduler
while read a minibatch from I_i **do**
    Pull w_j and V_j from server nodes for all features j that appear in this minibatch
    Compute the gradient based on (10.15) and (10.16)
    Push gradient back to servers
end while

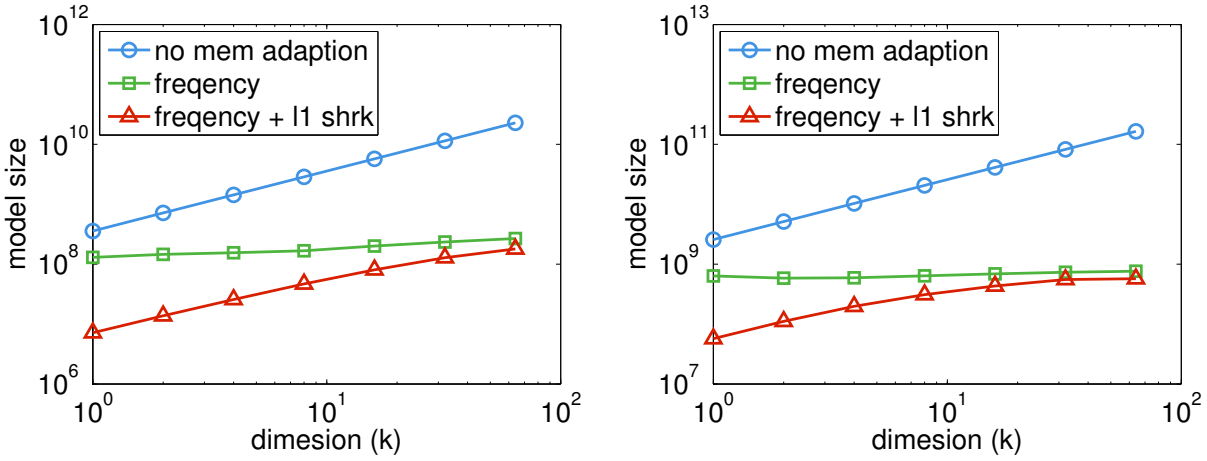
Server i:

if received gradient from a worker **then**
    update w and V by using (10.19) and (10.18)
end if

server node handles the model pull request from worker nodes. Upon receiving gradients from the worker nodes, the server node updates the model using the update rules discussed in Section 10.4.1.

Worker nodes perform most of the actual computation. After receiving a *process examples* I command from the scheduler, a worker repeatedly reads minibatches from I, which are files stored in a distributed file system. For each minibatch $B \subseteq I$, the worker first finds the supporting feature indices $J = \{j : x_{ij} \neq 0 \text{ for } i \in B\}$, then it pulls (prefetch) the working set of model parameters from the server nodes, namely $\{w_j, V_j : i \in J\}$, finally the worker node calculates the gradient of the minibatch and pushes them to the servers.

Note that a worker node only needs to cache the minibatches being processed and the associated working set of parameters to minimize the memory consumption. It also parallelizes data IO, computation, and communication to hide the IO cost. Besides, a worker node uses various filters to reduce the amount of data that needs to be communicated. We adopt standard filters provided



(a) Dataset: Criteo (b) Dataset: CTRa

Figure 10.1: Number of non-zero entries in V.

by the parameter server [83] including the key caching and data compression. In addition, we use fixed-point encoding, namely we convert the floating-point weight and gradient into shorter fixed-point integers during communication.

The codes of our implementation of DiFacto are public available in the DMLC project[1].

10.5 Experiments

We run DiFacto with 100 workers and 100 servers on 10 physical machines on Amazon EC2 EC2-c4.8x. We fix the dimension $k = 16$ and the minibatch size 10^4 for Criteo and 10^3 for CTRa. For smaller datasets, we decrease the minibatch sizes by 10 times respectively. We use a fixed regularizer for w with constants $\lambda_1 = 4$ and $\lambda = 0$, and we optimize the constants μ by searching within the range of $\{0, 10^{-5}, 10^{-4}, 10^{-3}\}$ based on an additional validation set. The elements in V were initialized uniformly at random in the range $[-0.01, 0.01]$. The fast learning rate η_w is selected between 0.001 and 0.1 with $\eta_V = \eta_w$ and $\beta_w = \beta_V = 1$. Finally, we terminate the algorithm when the objective value on the validation set stops decreasing.

10.5.1 Adaptive memory

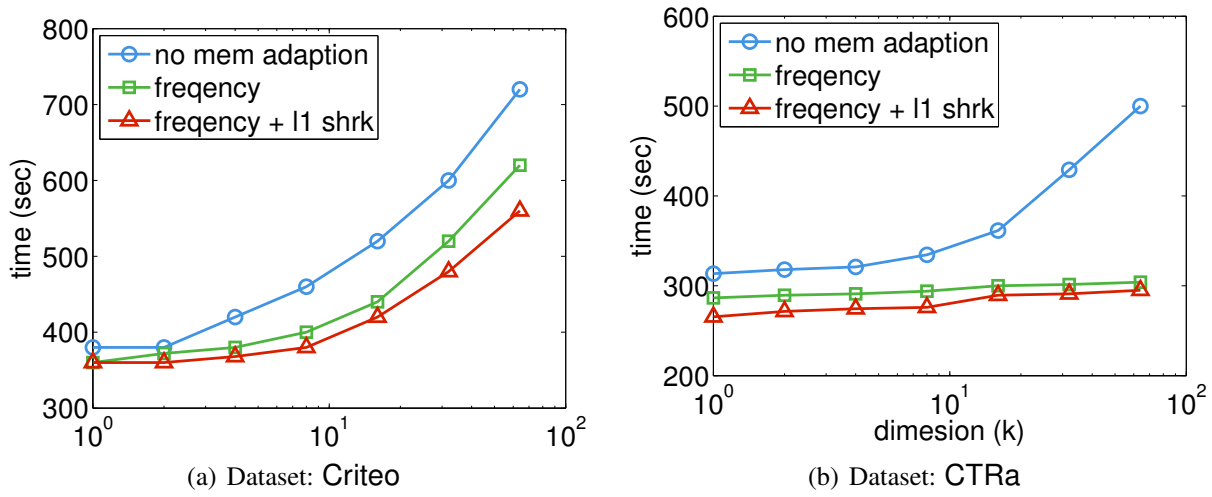
We first study the effectiveness of adaptive memory. We compare the following three settings which use different memory adaptive constraints in (10.6):

No memory adaption No constraint is applied, namely we do not force elements in V_i to 0.

Frequency threshold We only impose the constraint $V_i = 0$ for infrequent keys which have sparsity $n_i < k$. (recall that n_i is the occurrence of feature i in the data.)

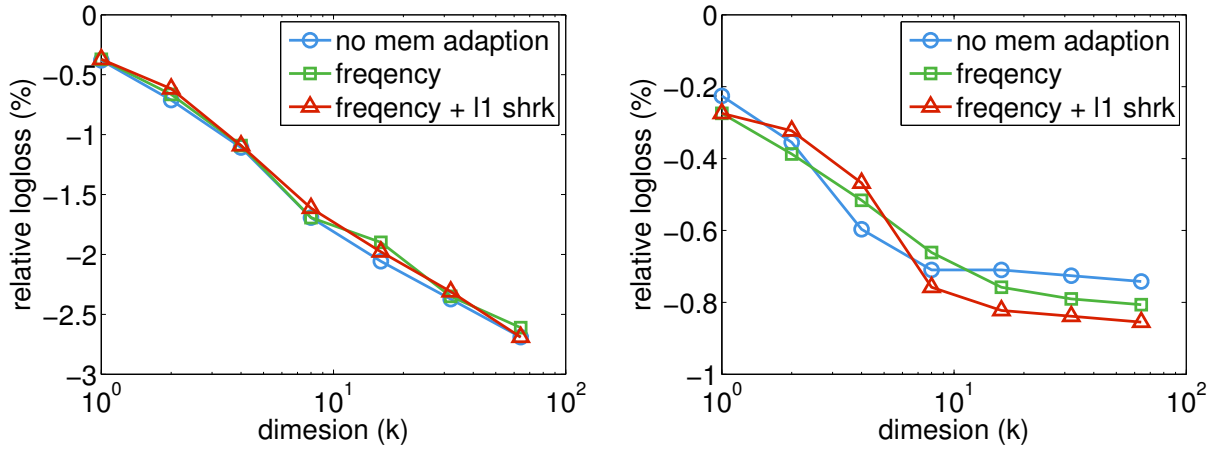
Frequency threshold + ℓ_1 shrinkage We impose the constraint $V_i = 0$ if $w_i = 0$. That is, we mark V_i as inactive if w_i is set to 0 by the sparse induction ℓ_1 regularization.

[1] `http://dmlc.github.io/`



(a) Dataset: Criteo (b) Dataset: CTRa

Figure 10.2: Runtime for one iteration.



(a) Dataset: Criteo (b) Dataset: CTRa

Figure 10.3: Relative test logloss compared to logistic regression ($k = 0$ and 0 relative loss).

We use the same hyper-parameters for the three settings and vary the dimension k from 1 to 64.

Figure 10.1 shows that the memory adaptive constraints effectively reduce the model size. The reduction is about 100x for Criteo and 300x for CTRa when $k = 64$. Figure 10.2 shows that they also brings about a 20% reduction in the run time. These constraints also significantly decrease the server node memory consumption and network traffic. Especially when k is large, the resulted amount of communication may overwhelm the system. Being able to reduce the model size benefits the system performance a lot. A more interesting observation from 10.3 is that these memory adaptive constraints do not affect the accuracy. To the contrary, we even see a slight improvement when the dimension k is greater than 8 for CTRa. The reason could be that the model capacity control is of great importance when the dimension k is large, and these memory adaptive constraints can provide additional capacity control besides the ℓ_2 and ℓ_1

regularizers.

10.5.2 Fixed-point Compression

We evaluate the lossy fixed-point compression for data communication. By default, both the model and gradient entries are represented as 32 bit floating numbers. In this experiment, we compress these values to lower precision integers. More specifically, given a bin size b and number of bits n, we represent x by the following n-bit integer

$$z := \left\lfloor \frac{x}{b} \times 2^n \right\rfloor + \sigma, \tag{10.23}$$

where σ is a Bernoulli random variable chosen such as to ensure that $\mathbf{E}[z] = 2^n x/b$. We implemented the fixed-point compression as a user-defined filter in the parameter server framework. Since multiple numbers are communicated in each round, we choose b to be the absolute maximum value of these numbers. In addition, we used the key caching and lossless data compression (via LZ4) filters.

The results for $n = 8, 16, 24$ are shown in Figure 10.4 and 10.5. As expected, fixed-point compression linearly reduces the volume of network traffic, which is dominated by communicating the model and gradient. We also observe that we can obtain a 4.2x compression rate from 32-bit floating-point to 8-bit fixed-point on Criteo. The reason is the latter improves the compression rate for the following lossless LZ4 compression.

We observe different effects of accuracy on these two datasets: CTRa is robust to the number precision, while Criteo has a 6% increase of logloss if only using 1-byte presentation. However, a medium compression rate even improves the model accuracy. The reason might be that the lossy compression acts as a regularization to the objective function.

10.5.3 Comparison with LibFM

To our best knowledge, there is no publicly released distributed FM solver. Hence we only compare DiFacto to the popular single machine package LibFM developed by Rendle [114]. We only report results on smaller datasets sampled from Criteo and CTRa on a single machine, since LibFM fails on the other two larger datasets. We perform a similar grid search of the hyper-parameters as we did for DiFacto. As LibFM only uses single thread, we run DiFacto with 1 worker and 1 server in sequential execution order. We also report the performance using 10 workers and 10 servers on a single machine for reference.

Figure 10.6 shows that DiFacto converges significantly faster than LibFM, it uses 2 times fewer iterations to reach the best model. This is because the adaptive learning rate used in DiFacto better models the data sparsity and the adaptive regularization and constraints can further accelerate the convergence. In particular, the latter results in a lower test logloss on the CTRa dataset, where the number of features exceeds the number of examples, requiring improved capacity control.

Also note that DiFacto with a single worker is twice slower than LibFM per iteration. This is because the data communication overhead between the worker and the server cannot be ignored in the sequential execution. More importantly, DiFacto does not require any data preprocessing

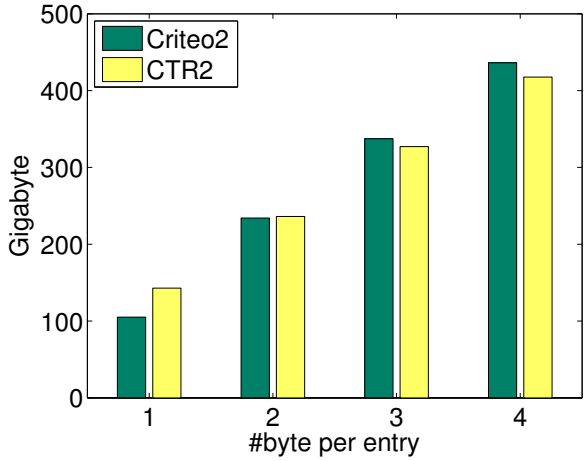


Figure 10.4: Total data sent by workers in one iteration. The compression rates from 4-byte to 1-byte are 4.2x and 2.9x for Criteo and CTRa, respectively.

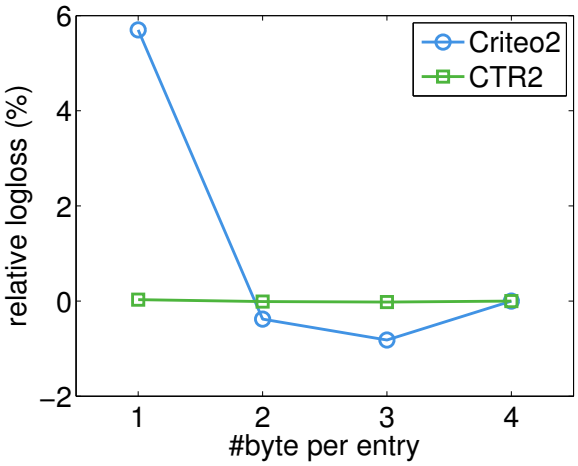
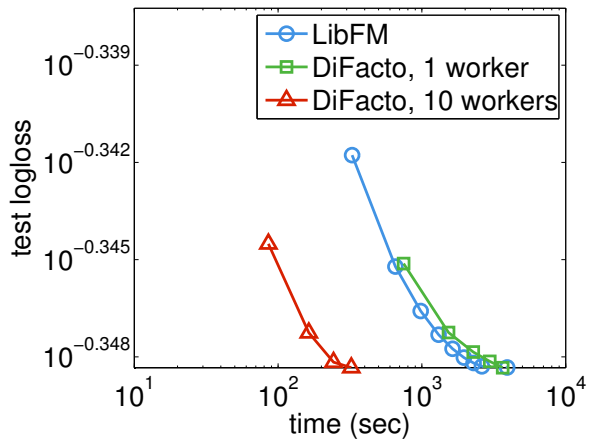
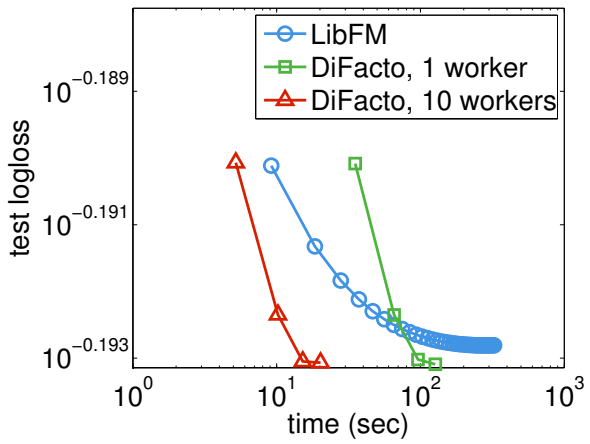


Figure 10.5: The relative test logloss compared to no fixed-point compression.



(a) 46 million examples from Criteo



(b) 0.3 million examples from CTRa

Figure 10.6: Comparison with LibFM on a single machine.

to map arbitrary 64-bit integer and string feature indices, which are used in both Criteo and CTRa, to continuous integer indices. The cost of this data preprocessing step, required by LibFM but not shown in Figure 10.6, even exceeds the cost of the actual training process (1,400 seconds for Criteo). Nevertheless, DiFacto with a single worker still outperforms LibFM. Moreoverx, it is 10 times faster than LibFM when using 10 workers on the same machine.

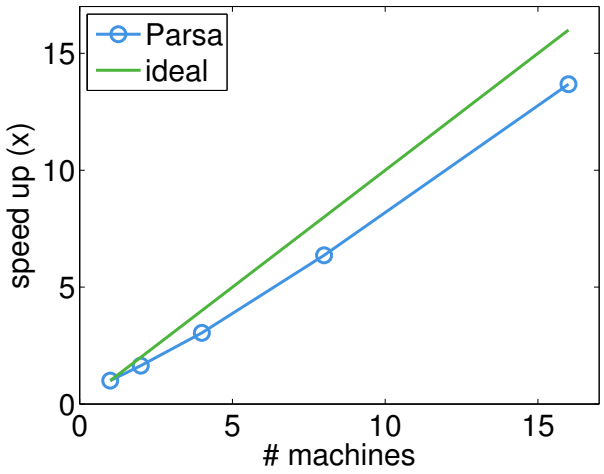


Figure 10.7: The speedup from 1 machine to 16 machines, where each machine runs 10 workers and 10 servers.

10.5.4 Scalability

Finally, we study the scalability of DiFacto by varying the number of physical machines used in training. We assign 10 worker nodes and 10 server nodes to each machine, and increase the number of machines from 1 to 16. Figure 10.7 shows that DiFacto achieves an 8x speedup when the number of machines increases from to 16 for both Criteo and CTRa. The reason for the satisfactory performance is twofold. First, asynchronous SGD eliminates the need for synchronization between workers and it has a high tolerance for stragglers. Even though we used dedicated machines, they still share network bandwidth with others. In particular, we observe a large variation of read speed when streaming data from Amazon's S3 service, despite using the IO optimized c4.8xlarge series of machines. Second, DiFacto uses several filters which effectively reduce the amount of network traffic. We observe that even though CTRa produces 10 times more network traffic than Criteo, they have similar speedup performance.

There is a longstanding suspicion that the convergence of asynchronous SGD slows down when the number of workers increases. Nonetheless, we do not observe a substantial difference in model accuracy. In other words, the relative difference in the objective logloss on test datasets is below 0.5% when increasing the number of workers from 10 to 160. The reason might be that the datasets we use are highly sparse and the features are not extremely correlated, and hence the inconsistency due to concurrently updating by multiple workers may not have a major effect. These observations are consistent with our convergence analysis of the algorithm.

Chapter 11

Conclusion

Bibliography

[1] Geforce 10 series. https://en.wikipedia.org/wiki/GeForce_10_series. 2.1

[2] Detailed specifications of the intel xeon e5-2600v4 broadwell-ep processors. https://www.microway.com/knowledge-center-articles/detailed-specifications-of-the-intel-xeon-e5-2600v4-broadwell-ep-processors/. 2.1

[3] Top500 lists, 2016. https://www.top500.org/lists/2016/06/. 2.1

[4] A. Agarwal and J. C. Duchi. Distributed delayed stochastic optimization. In *IEEE Conference on Decision and Control*, pages 5451–5452. IEEE, 2012. 8.1, 8.2.1, 8.2.3, 8.3.1

[5] A. Ahmed, Y. Low, M. Aly, V. Josifovski, and A. J. Smola. Scalable inference of dynamic user interests for behavioural targeting. In *Knowledge Discovery and Data Mining*, 2011. 3.4.2

[6] A. Ahmed, Mohamed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *Proceedings of The 5th ACM International Conference on Web Search and Data Mining (WSDM)*, 2012. 3.1.3, 3.4.2, 9.1

[7] A. Ahmed, Nino Shervashidze, Shravan Narayanamurthy, Vanja Josifovski, and Alexander J. Smola. Distributed large-scale natural graph factorization. In *World Wide Web Conference*, Rio de Janeiro, 2013. 6.1, 9.1, 9.1, 9.2, 10.1

[8] Amazon. Amazon web services. https://aws.amazon.com/. 1, 1.1.2

[9] R. Andersen, D.F. Gleich, and V.S. Mirrokni. Overlapping clusters for distributed computation. In E. Adar, J. Teevan, E. Agichtein, and Y. Maarek, editors, *Proceedings of the Fifth International Conference on Web Search and Web Data Mining, WSDM 2012, Seattle, WA, USA, February 8-12, 2012*, pages 273–282. ACM, 2012. URL `http://doi.acm.org/10.1145/2124295.2124330`. 9.2

[10] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, second edition, 1995. 3.2.1

[11] Galen Andrew and Jianfeng Gao. Scalable training of l_1-regularized log-linear models. In *Proceedings of the International Conference on Machine Learning*, pages 33–40, New York, NY, USA, 2007. ACM. 6.3.1

[12] Apache Foundation. Mahout project, 2012. `http://mahout.apache.org`. 3.1.3

[13] L. A. Barroso and H. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 4(1):1–108,

2009. 2.2, 2.2

[14] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. Theano: new features and speed improvements. *arXiv preprint arXiv:1211.5590*, 2012. 4.1.1, 4.1.2

[15] R. M. Bell and Y. Koren. Lessons from the netflix prize challenge. *SIGKDD Explorations*, 9(2):75–79, 2007. URL `http://doi.acm.org/10.1145/1345448.1345465`. 10.2.1

[16] R. Berinde, G. Cormode, P. Indyk, and M.J. Strauss. Space-optimal heavy hitters with strong error bounds. In J. Paredaens and J. Su, editors, *Proceedings of the Twenty-Eigth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS*, pages 157–166. ACM, 2009. URL `http://doi.acm.org/10.1145/1559795.1559819`. 3.4.3

[17] D. Bertsekas and J. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, 1989. 5

[18] J. Besag. Spatial interaction and the statistical analysis of lattice systems (with discussion). *Journal of the Royal Statistical Society. Series B*, 36(2):192–236, 1974. 9.2

[19] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *arXiv preprint arXiv:1606.04838*, 2016. 5

[20] Florian Bourse, Marc Lelarge, and Milan Vojnovic. Balanced graph edge partition. In *ACM KDD 2014*, August 2014. 9.1

[21] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–123, 2010. 6.3.2

[22] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, Cambridge, England, 2004. 5

[23] J.K. Bradley, A. Kyrola, D. Bickson, and C. Guestrin. Parallel coordinate descent for L1-regularized loss minimization. In L. Getoor and T. Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning*, pages 321–328. Omnipress, 2011. 6.1, 6.2.2, 6.3.1

[24] J. Byers, J. Considine, and M. Mitzenmacher. Simple load balancing for distributed hash tables. In *Peer-to-peer systems II*, pages 80–87. Springer, 2003. 3.3.3

[25] RH Byrd, SL Hansen, Jorge Nocedal, and Y Singer. A stochastic quasi-newton method for large-scale optimization. *arXiv preprint arXiv:1401.7020*, 2014. 7.1.3

[26] Richard H Byrd, Gillian M Chin, Jorge Nocedal, and Yuchen Wu. Sample size selection in optimization methods for machine learning. *Mathematical programming*, 134(1):127–155, 2012. 7.1.2

[27] K. Canini. Sibyl: A system for large scale supervised machine learning. *Technical Talk*, 2012. URL `http://users.soe.ucsc.edu/~niejiazhong/slides/chandra.pdf`. 1, 1.1.1, 3.4.1

[28] U.V. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999. 9.1, 9.1, 9.3

[29] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association. 2.2

[30] Kai-Wei Chang and Dan Roth. Selective block minimization for faster convergence of limited memory large-scale linear models. In *Conference on Knowledge Discovery and Data Mining*, pages 699–707, 2011. 7.1.3

[31] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015. 1.3, 4.3.2, 4.5

[32] Byung-Gon Chun, Tyson Condie, Carlo Curino, Chris Douglas, Sergiy Matusevych, Brandon Myers, Shravan Narayanamurthy, Raghu Ramakrishnan, Sriram Rao, Josh Rosen, et al. Reef: Retainable evaluator execution framework. *Proceedings of the VLDB Endowment*, 6(12):1370–1373, 2013. 3.1.2

[33] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011. 4.1.1, 4.1.2

[34] P. L. Combettes and J. C. Pesquet. Proximal splitting methods in signal processing. In *Fixed-Point Algorithms for Inverse Problems in Science and Engineering*, pages 185–212. Springer, 2011. 5.1

[35] The Kubernetes Community. Kubernetes: Production-grade container orchestration, 2016. http://kubernetes.io. 2.2

[36] Minerva contributors. Minerva: a fast and flexible system for deep learning. https://github.com/dmlc/minerva. 4.6

[37] G. Cormode and S. Muthukrishnan. Summarizing and mining skewed data streams. In *SDM*, 2005. 3.4.3, 3.4.3

[38] Andrew Cotter, Ohad Shamir, Nati Srebro, and Karthik Sridharan. Better mini-batch algorithms via accelerated gradient methods. In *NIPS*, volume 24, pages 1647–1655, 2011. 7.1.3

[39] Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, Soren Lassen, Philip Pronin, Sriram Sankar, et al. Unicorn: A system for searching the social graph. *VLDB*, 6(11):1150–1161, 2013. 9.1

[40] Wei Dai, Jinliang Wei, Xun Zheng, Jin Kyu Kim, Seunghak Lee, Junming Yin, Qirong Ho, and Eric P Xing. Petuum: A framework for iterative-convergent distributed ml. *arXiv preprint arXiv:1312.7651*, 2013. 3.1.2, 3.2.1

[41] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *CACM*, 51(1):107–113, 2008. URL http://doi.acm.org/10.1145/1327452.1327492. 1.1.2, 2.2, 3.1.3

[42] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Ng. Large scale distributed deep networks. In *Neural Information Processing Systems*, 2012. 3.1.3, 3.2, 6.3.2, 10.1

[43] 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. In T. C. Bressoud and M. F. Kaashoek, editors, *Symposium on Operating Systems Principles*, pages 205–220. ACM, 2007. ISBN 978-1-59593-591-5. URL http://doi.acm.org/10.1145/1294261.1294281. 2.2, 3.3.1

[44] Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. Optimal distributed online prediction using mini-batches. Technical report, http://arxiv.org/abs/1012.1367, 2010. 7.1.2, 7.1.3, 7.1.4

[45] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A large-scale hierarchical image database. In *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, 2009. 4.5.3

[46] CXXNet Developers. Cxxnet: fast, concise, distributed deep learning framework, 2015. https://github.com/dmlc/cxxnet. 4.6

[47] K.D. Devine, E.G. Boman, R.T. Heaphy, R.H. Bisseling, and U.V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Parallel and Distributed Processing Symposium*, pages 10–pp. IEEE, 2006. 9.1, 9.1

[48] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:18–32, 1988. 3.2.1

[49] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2010. 10.4.1

[50] John Duchi, Michael I Jordan, and Brendan McMahan. Estimation, optimization, and parallelism when data is sparse. In *NIPS 26*, pages 2832–2840, 2013. 8.1

[51] R.-E. Fan, J.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, August 2008. 6.3.1, 7.3

[52] The Apache Software Foundation. Apache hadoop nextgen mapreduce (yarn). http://hadoop.apache.org/. 2.2, 3.3.5

[53] The Apache Software Foundation. Apache hadoop, 2009. http://hadoop.apache.org/core/. 2.2, 3.1.3

[54] The Apache Software Foundation. Cassandra: Manage massive amounts of data, fast, without losing sleep, 2016. http://cassandra.apache.org/. 2.2

[55] The Apache Software Foundation. Flink: open source platform for distributed stream and

batch data processing, 2016. http://flink.apache.org/. 2.2

[56] Saeed Ghadimi and Guanghui Lan. Stochastic first-and zeroth-order methods for nonconvex stochastic programming. *SIAM Journal on Optimization*, 23(4):2341–2368, 2013. 10.4.2, 10.4.2

[57] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Operating Systems Review*, 37(5):29–43, 2003. 2.2

[58] Kevin Gimpel, Dipanjan Das, and Noah A Smith. Distributed asynchronous online learning for natural language processing. In *Proceedings of the Fourteenth Conference on Computational Natural Language Learning*, pages 213–222. Association for Computational Linguistics, 2010. 7.1.3

[59] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2012)*, 2012. 9.1, 9.1, 9.2

[60] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, 2014. 9.1

[61] Google. Google cloud. https://cloud.google.com/. 1, 1.1.2

[62] T.L. Griffiths and M. Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences*, 101:5228–5235, 2004. 3.4.2

[63] Steinar H. Gunderson. Snappy: A fast compressor/decompressor. `https://code.google.com/p/snappy/`. 3.3.2

[64] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL `http://arxiv.org/abs/1512.03385`. 1, 1.1.1

[65] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. *CoRR*, abs/1603.05027, 2016. URL `http://arxiv.org/abs/1603.05027`. 4.5, 4.5.3

[66] I. Heller and C. Tompkins. An extension of a theorem of dantzig's. In H.W. Kuhn and A.W. Tucker, editors, *Linear Inequalities and Related Systems*, volume 38 of *Annals of Mathematics Studies*. AMS, 1956. 9.3.2

[67] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 22–22, 2011. 2.2, 3.3.5

[68] Q. Ho, J. Cipar, H. Cui, S. Lee, J. Kim, P. Gibbons, G. Gibson, G. Ganger, and E. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*, 2013. 3.1.3

[69] M. Hoffman, D. M. Blei, C. Wang, and J. Paisley. Stochastic variational inference. In *International Conference on Machine Learning*, 2012. 3.4.2

[70] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pages 675–678. ACM, 2014. 4.1.1, 4.1.2

[71] T. Joachims. Making large-scale SVM learning practical. In B. Schölkopf, C. J. C. Burges, and A. J. Smola, editors, *Advances in Kernel Methods — Support Vector Learning*, pages 169–184, Cambridge, MA, 1999. MIT Press. 6.3.1

[72] Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in Neural Information Processing Systems*, pages 315–323, 2013. 7.1.3

[73] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1), 1998. 9.1, 9.1

[74] Larry Kim. How many ads does google serve in a day?, 2012. URL `http://goo.gl/oIidXO`. `http://goo.gl/oIidXO`. 1.1.1

[75] Tim Kraska, Ameet Talwalkar, John C Duchi, Rean Griffith, Michael J Franklin, and Michael I Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, 2013. 3.1.2, 3.2.1

[76] Frank Kschischang, Brendan J. Frey, and Hans-Andrea Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, 2001. 9.2

[77] Brian Kulis and Peter L Bartlett. Implicit online learning. In *Proc. Intl. Conf. Machine Learning*, 2010. 7.1.3

[78] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001. 3.3.1

[79] J. Langford, A. J. Smola, and M. Zinkevich. Slow learners are fast. In *Neural Information Processing Systems*, 2009. URL `http://arxiv.org/abs/0911.0491`. 6.1, 8.1

[80] Q.V. Le, A. Karpenko, J. Ngiam, and A.Y. Ng. ICA with reconstruction cost for efficient overcomplete feature learning. *Advances in Neural Information Processing Systems*, 24: 1017–1025, 2011. 6.1, 6.3.2

[81] Q.V. Le, T. Sarlos, and A. J. Smola. Fastfood — computing hilbert space expansions in loglinear time. In *International Conference on Machine Learning*, 2013. 10.1

[82] M. Li, D. G. Andersen, and A. J. Smola. Distributed delayed proximal gradient methods. In *NIPS Workshop on Optimization for Machine Learning*, 2013. 3.1.3

[83] M. Li, D. G. Andersen, J. Park, A. J. Smola, A. Amhed, V. Josifovski, J. Long, E. Shekita, and B. Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014. 1.3, 10.4.3

[84] M. Li, D. G. Andersen, A. J. Smola, and K. Yu. Communication efficient distributed machine learning with the parameter server. In *Neural Information Processing Systems*, 2014. 1.3, 10.4.2

[85] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient mini-batch train-

ing for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–670. ACM, 2014. 1.3

[86] Mu Li, Dave G Andersen, and Alexander J Smola. Graph partitioning via parallel submodular approximation to accelerate distributed machine learning. *arXiv preprint arXiv:1505.04636*, 2015. 1.3

[87] Mu Li, Ziqi Liu, Alexander J Smola, and Yu-Xiang Wang. Difacto: Distributed factorization machines. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, pages 377–386. ACM, 2016. 1.3

[88] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. *arXiv preprint arXiv:1506.08272*, 2015. 10.4.2, 24, 25, 10.4.2

[89] Dong C. Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(3):503–528, 1989. 5.1, 6.3.1, 7.3, 7.3

[90] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence*, 2010. URL `https://select.cs.cmu.edu/code/graphlab/index.html`. 9.1

[91] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. In *PVLDB*, 2012. 3.1.2, 3.1.3

[92] David G Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee, 1999. 1.1.1

[93] Dhruv Mahajan, S Sathiya Keerthi, S Sundararajan, and Leon Bottou. A parallel sgd method with strong convergence. *arXiv preprint arXiv:1311.0636*, 2013. 7.1.3

[94] G. Mann, R. McDonald, M. Mohri, N. Silberman, and D. Walker. Efficient large-scale distributed training of conditional maximum entropy models. In Y. Bengio, D. Schuurmans, J. Lafferty, C. K. I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems 22*, pages 1231–1239, 2009. 7.1.3

[95] Abadi Martn, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous systems. 2015. 4.1.1, 4.1.2

[96] S. Matsushima, S.V.N. Vishwanathan, and A.J. Smola. Linear support vector machines via dual cached loops. In Q. Yang, D. Agarwal, and J. Pei, editors, *The 18th ACM SIGKDD In-*

ternational Conference on Knowledge Discovery and Data Mining, KDD, pages 177–185. ACM, 2012. URL http://dl.acm.org/citation.cfm?id=2339530. 6.3.1, 7.1.3

[97] B. McMahan. Follow-the-regularized-leader and mirror descent: Equivalence theorems and l1 regularization. In *International Conference on Artificial Intelligence and Statistics*, pages 525–533, 2011. 10.4.1

[98] Brendan McMahan and Matthew Streeter. Delay-tolerant algorithms for asynchronous distributed online learning. In *Advances in Neural Information Processing Systems*, pages 2915–2923, 2014. 8.1, 8.3.1, 8.3.2, 8.3.2

[99] H Brendan McMahan, Gary Holt, D Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, and Daniel Golovin. Ad click prediction: a view from the trenches. In *KDD*, 2013. 1.1.1, 10.2.2, 10.3.2

[100] Microsoft. Microsfot azure. https://azure.microsoft.com/. 1, 1.1.2

[101] T. Moon, A. J. Smola, Y. Chang, and Z. Zheng. Intervalrank: isotonic regression with listwise and pairwise constraints. In B.D. Davison, T. Suel, N. Craswell, and B. Liu, editors, *Proceedings of the Third International Conference on Web Search and Web Data Mining, WSDM*, pages 151–160. ACM, 2010. 10.2.1

[102] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013. 3.1.2, 3.2.1

[103] A Nedić, Dimitri P Bertsekas, and Vivek S Borkar. Distributed asynchronous incremental subgradient methods. *Studies in Computational Mathematics*, 8:381–407, 2001. 8.1

[104] S. Negahban, P. Ravikumar, M.J. Wainwright, and B. Yu. A unified framework for high-dimensional analysis of m-estimators with decomposable regularizers. *arXiv preprint arXiv:1010.2731*, 2010. 10.3.2

[105] Joel Nishimura and Johan Ugander. Restreaming graph partitioning: Simple versatile algorithms for advanced balancing. In *ACM SIGKDD*, pages 1106–1114. ACM, 2013. 9.1

[106] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer Series in Operations Research. Springer, 2nd edition, 2006. 5

[107] J.B. Orlin. A faster strongly polynomial time algorithm for submodular function minimization. *Mathematical Programming*, 118(2):237–251, 2009. 9.4.1

[108] N. Parikh and S. Boyd. Proximal algorithms. *To appear in Foundations and Trends in Optimization*, 2013. 5.1, 6.1

[109] K. B. Petersen and M. S. Pedersen. The matrix cookbook, 2008. URL http://www2.imm.dtu.dk/pubdb/p.php?3274. Version 20081110. 6.3.2

[110] Amar Phanishayee, David G Andersen, Himabindu Pucha, Anna Povzner, and Wendy Belluomini. Flex-kv: Enabling high-performance and flexible kv systems. In *Proceedings of the 2012 workshop on Management of big data systems*, pages 19–24. ACM, 2012. 2

[111] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In R. H. Arpaci-Dusseau and B. Chen, editors, *Operating Systems Design and Implementation, OSDI*, pages 293–306. USENIX Association, 2010. URL `http://www.usenix.org/event/osdi10/tech/full_papers/osdi10_proceedings.pdf`. 3.1.3

[112] Josep M Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. The little engine (s) that could: scaling online social networks. *ACM SIGCOMM Computer Communication Review*, 41(4):375–386, 2011. 9.1

[113] B. Recht, C. Re, S.J. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In Peter Bartlett, Fernando Pereira, Richard Zemel, John Shawe-Taylor, and Kilian Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 693–701, 2011. URL `http://books.nips.cc/nips24.html`. 6.1, 6.2.2

[114] S. Rendle and L. Schmidt-Thieme. Pairwise interaction tensor factorization for personalized tag recommendation. In *WSDM '10: Proceedings of the third ACM international conference on Web search and data mining*, pages 81–90. ACM, 2010. doi: http://doi.acm.org/10.1145/1718487.1718498. 10.1, 10.2.2, 10.5.3

[115] Steffen Rendle. Time-Variant Factorization Models Context-Aware Ranking with Factorization Models. volume 330 of *Studies in Computational Intelligence*, chapter 9, pages 137–153. 2011. ISBN 978-3-642-16897-0. 10.1, 10.2.2

[116] P. Richtárik and M. Takáč. Iteration complexity of randomized block-coordinate descent methods for minimizing a composite function. *Mathematical Programming*, pages 1–38, 2012. 6.1

[117] H. Robbins and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 1951. 1.1.3

[118] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, November 2001. 3.3.3

[119] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y. 4.1.1

[120] Shai Shalev-Shwartz and Tong Zhang. Accelerated mini-batch stochastic dual coordinate ascent. In *Advances in Neural Information Processing Systems*, pages 378–385, 2013. 7.1.3

[121] Amar Shan. Heterogeneous processing: a strategy for augmenting moore's law. *Linux Journal*, 2006(142):7, 2006. 2.1

[122] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory

cloud. In *ACM SIGMOD*, pages 505–516. ACM, 2013. 9.1

[123] A. J. Smola and S. Narayanamurthy. An architecture for parallel topic models. In *Very Large Databases (VLDB)*, 2010. 3.1, 3.1.3, 3.2.5, 9.1

[124] E. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska. Mli: An api for distributed machine learning. 2013. 3.1.3

[125] S. Sra. Scalable nonconvex inexact proximal splitting. In *Neural Information Processing Systems*, 2012. 6.1

[126] Suvrit Sra, Sebastian Nowozin, and Stephen J Wright. *Optimization for machine learning*. 2012. 5

[127] Suvrit Sra, Adams Wei Yu, Mu Li, and Alexander J Smola. Adadelay: Delay adaptive distributed stochastic convex optimization. *arXiv preprint arXiv:1508.05003*, 2015. 1.3, 8.2.3

[128] N. Srebro, N. Alon, and T. Jaakkola. Generalization error bounds for collaborative prediction with low-rank matrices. In L. K. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems 17*, Cambridge, MA, 2005. MIT Press. 10.3.3

[129] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014. 6.2.2

[130] I. Stanton. Streaming balanced graph partitioning for random graphs. *CoRR*, abs/1212.1121, 2012. 9.1

[131] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230. ACM, 2012. 9.1, 9.1

[132] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001. 3.3

[133] Z. Svitkina and L. Fleischer. Submodular approximation: Sampling-based algorithms and lower bounds. *SIAM Journal on Computing*, 40(6):1715–1737, 2011. 9.3.1, 9.3.1

[134] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014. URL `http://arxiv.org/abs/1409.4842`. 1.1.1

[135] Martin Takáč, Avleen Bijral, Peter Richtárik, and Nathan Srebro. Mini-batch primal and dual methods for svms. *arXiv preprint arXiv:1303.2314*, 2013. 7.1.3

[136] B. Taskar, C. Guestrin, and D. Koller. Max-margin Markov networks. In S. Thrun, L. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16*, pages 25–32, Cambridge, MA, 2004. MIT Press. 10.2.1

[137] Christina Teflioudi, Faraz Makari, and Rainer Gemulla. Distributed matrix completion. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*, pages 655–664.

IEEE, 2012. 6.1

[138] Choon Hui Teo, S. V. N. Vishwanthan, A. J. Smola, and Quoc V. Le. Bundle methods for regularized risk minimization. *Journal of Machine Learning Research*, 11:311–365, January 2010. 6.1

[139] R. Tibshirani. Regression shrinkage and selection via the lasso. *J. R. Stat. Soc. Ser. B Stat. Methodol.*, 58:267–288, 1996. 10.3.2

[140] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *WSDM*, pages 333–342. ACM, 2014. 9.1

[141] Johan Ugander and Lars Backstrom. Balanced label propagation for partitioning massive graphs. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 507–516. ACM, 2013. 9.1

[142] Robbert van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004. 3.3

[143] V. Vapnik. *Statistical Learning Theory*. John Wiley and Sons, New York, 1998. 1.1.3

[144] Venkateshwaran Venkataramani, Zach Amsden, Nathan Bronson, George Cabrera III, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Jeremy Hoon, et al. Tao: how facebook serves the social graph. In *ACM SIGMOD*, pages 791–792. ACM, 2012. 9.1

[145] G. Wahba. *Spline Models for Observational Data*, volume 59 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. SIAM, Philadelphia, 1990. 10.3.2

[146] R.C. Whaley, A. Petitet, and J.J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. 3.2.1

[147] Shengqi Yang, Xifeng Yan, Bo Zong, and Arijit Khan. Towards effective partition management for large graphs. In *ACM SIGMOD*, pages 517–528, 2012. 9.1

[148] J. Ye, J.-H. Chow, J. Chen, and Z. Zheng. Stochastic gradient boosted distributed decision trees. In D. W.-L. Cheung, I.-Y. Song, W.W. Chu, X. Hu, and J.J. Lin, editors, *Conference on Information and Knowledge Management, CIKM*, pages 2061–2064. ACM, 2009. URL `http://doi.acm.org/10.1145/1645953.1646301`. 10.1

[149] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Zhiheng Huang, Brian Guenter, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Huaming Wang, et al. An introduction to computational networks and the computational network toolkit. Technical report, Technical report, Tech. Rep. MSR, Microsoft Research, 2014, 2014. research. microsoft. com/apps/pubs, 2014. 4.1.2

[150] H.-F. Yu, C.-J. Hsieh, K.-W. Chang, and C.-J. Lin. Large linear classification when data cannot fit in memory. In B. Rao, B. Krishnapuram, A. Tomkins, and Q. Yang, editors, *Knowledge Discovery and Data Mining*, pages 833–842. ACM, 2010. URL `http://doi.acm.org/10.1145/1835804.1835910`. 7.1.2, 7.1.3, 7.2.3

[151] G. X. Yuan, K. W. Chang, C. J. Hsieh, and C. J. Lin. A comparison of optimization methods and software for large-scale l1-regularized linear classification. *Journal of Machine*

Learning Research, pages 3183–3234, 2010. 6.1

[152] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *HotCloud 2010*, June 2010. 2.2

[153] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy Mccauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Fast and interactive analytics over hadoop data with spark. *USENIX ;login:*, 37(4):45–51, August 2012. 3.1.3

[154] Tong Zhang and Frank J. Oles. Text categorization based on regularized linear classification methods. *Information Retrieval*, 4:5–31, 2001. 6.3.1

[155] Jingren Zhou, Nicolas Bruno, and Wei Lin. Advanced partitioning techniques for massively distributed computation. In *ACM SIGMOD*, pages 13–24, 2012. 9.1

[156] M. Zinkevich. Online convex programming and generalised infinitesimal gradient ascent. In *Proceedings of the International Conference on Machine Learning*, pages 928–936, 2003. 7.1.2

[157] Martin Zinkevich, A. J. Smola, Markus Weimer, and Lihong Li. Parallelized stochastic gradient descent. In nips23e, editor, *nips23*, pages 2595–2603, 2010. 6.1, 7.1.3