Scheduling for
Efficient Large-Scale Machine Learning Training

Jinliang Wei

CMU-CS-19-135
December 11, 2019

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Garth A. Gibson, Co-chair
Eric P. Xing, Co-chair
Phillip B. Gibbons
Gregory R. Ganger
Vijay Vasudevan, Google Brain

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

Copyright © 2019 Jinliang Wei

This research was sponsored by the National Science Foundation under grant numbers CCF-1629559 and IIS-1617583, Intel ISTC-CC, and the Defense Advanced Research Projects Agency under grant numbers FA8721-05-C-0003 and FA8702-15-D-0002. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.
**Keywords:** machine learning, distributed computing, distributed shared memory, static analysis, parallelization, scheduling, deep learning
To my family.
Abstract

Thanks to the rise and maturity of “Big Data” technology, the availability of big datasets on distributed computing systems attracts both academic researchers and industrial practitioners to apply more and more sophisticated machine learning techniques on those data to create higher value. Machine learning training, which summarizes the insights from big datasets as a mathematical model, is an essential step in any machine learning application. Due to the growing data size and model complexity, machine learning training demands increasingly high compute power and memory. In this dissertation, I present techniques that schedule computing tasks to utilize network bandwidth, computation, and memory better to improve training time and scale model size.

Machine learning training searches for optimal parameter values that maximize or minimize a particular objective function by repetitively processing the training dataset to refine these parameters in small steps. During this process, properly bounded error can be tolerated by performing extra search steps. Bounded error tolerance allows trading off learning progress for higher computation throughput, for example, by parallelizations that violate sequential semantics, and such trade-offs should be made carefully. Widely used machine learning frameworks, such as TensorFlow, represent the complex computation of a search step as a dataflow graph to enable global optimizations, such as operator fusion, data layout transformation, and dead code elimination. This dissertation leverages bounded error tolerance and the intermediate representation of training computation, such as dataflow graphs, to improve training efficiency.

First, I present a communication scheduling mechanism for data-parallel training that performs fine-grained communication and value-based prioritization for model parameters to reduce inconsistency in parameter values for faster convergence. Second, I present an automated computation scheduling mechanism that executes independent update computations in parallel with minimal programmer effort. Communication scheduling achieves faster convergence for data-parallel training, and when applicable, computation scheduling achieves even faster convergence using much less network bandwidth. Third, I present a mechanism to schedule computation and memory allocation based on the training computation’s dataflow graph to reduce GPU memory consumption and enable training much larger models without additional hardware.
Acknowledgments

In the past seven years, there were many occasions when I felt extremely grateful and it was beyond words to express my gratitude. I wish this section repays a tiny bit of all the kindness that I received.

First and foremost, I thank my advisors Garth Gibson and Eric Xing, who introduced me to machine learning systems research and supported me all the way. Garth patiently guided me through each step of my research and my graduate study and showed me how to be a great researcher by example. Our research discussions range from high-level directions to implementation and evaluation details. I am always amazed by how quickly Garth can get to the core of a problem and how broadly and deeply he knows about computer science. While I am still not proud of my speaking and writing skills, I would be nowhere near what I am today if Garth had not helped me practice my talks and revise my papers.

It was Eric who led me into the world of machine learning. Eric’s exceptional vision played a vital role in revealing the problems solved in this thesis. I thank Eric for his hand-by-hand guidance when I was young and for giving me the freedom to pursue my research goals when I became more mature. Eric’s persistent encouragement gave me the confidence to pursue higher goals.

I am grateful that Greg Ganger, Phil Gibbons, and Vijay Vasudevan joined my thesis committee. As members of the BigLearning group, Greg and Phil have been my mentor and collaborator since the early days of my graduate study. My research, especially Bösen and Orion, greatly benefited from their insights and advices. Additionally, I thank Greg for his great leadership of the Parallel Data Lab and I thank Phil for helping me with my job search.

Despite that I only became to know Vijay last year, Vijay is as kind and helpful to me as any thesis committee member can be. He is always responsive to my requests and gave me timely advices when I needed them, which could be ideas that improve my research, related work that I hadn’t known, and even pointers to code examples. I also thank Vijay for his many advices and tremendous help on my job search. I would not have been able to get the job that I am most excited for if it were not for Vijay’s help. I am thrilled to join an organization that Vijay is a part of.

Many other collaborators contributed to the research presented in this thesis. Wei (David) Dai is not only a valuable collaborator but also a close friend. I spent many days and nights discussing Bösen’s design with David and David helped implement the first several machine learning applications on Bösen. David is the first person that I go to when I need help with machine learning problems. I wish him a great success in pursuing entrepreneuship. Bösen greatly benefited from my collaboration with Henggang in the first couple of years of my graduate school. Henggang is one of the most disciplined students that I know and I learned a lot from him both on technical subjects and on personal discipline. Aurick helped me run Bösen experiments and implemented Virtual Mesh-
I am grateful to have him as a collaborator and as a friend. I met Anand Jayarajan when I visited Vector Institute at Toronto in the summer of 2018. As a junior graduate student, Anand impressed me with his passion for systems research and his hardworking attitude. I thank Anand for helping me run experiments with TensorFlowMem and wish him a successful career.

I thank other members of the BigLearning project, who helped me do better research and become a better person. Aaron Harlap inspired me with many interesting ideas and is fun to be around. I felt lucky to have stayed in the same hotel as Aaron when we were both interns at Microsoft Research in 2017. I thus had some quite interesting experiences, thanks to his company. I wish our friendship lasts despite my persistent refusal to trying weed. As a junior graduate student, I learned from Jin Kyu Kim on various technical subjects, the computer industry, and Korean culture. It was comforting to have Jin as a friend. I thank James Cipar and Qirong Ho for inventing SSP, which started the BigLearning project. I admire Abutalib Aghayev for his system hacking and C++ programming skills and his persistence in pursuing academic goals. I thank him for helping me revise my paper on Orion and I look forward to calling him Professor soon.

I was fortunate to be part of the Parallel Data Lab (PDL). PDL gave me the opportunity to interact with the broad systems research community at CMU. PDL provides several compute clusters, which are my major experimental platform. The weekly PDL meetings gave me a break from my daily research, and thanks to those meetings, I got to eat more fruits and am able to better understand and appreciate other system research topics. The annual retreats and visit days pushed me to polish my presentation skills, and I appreciate the interaction with industrial attendees. I thank Karen Lindenfelser and Bill Courtright for organizing PDL events and keep PDL functioning, and I thank Joan Digney for helping me create nice-looking posters and never complaining no matter how late I send her my drafts. I thank Mitch Franzos, Chuck Cranor, Jason Boles, Chad Dougherty, Zisimos Economou, and Charlene Zang for maintaining the PDL clusters and helping me with many questions.

I thank the faculty and students who were active in PDL for creating a friendly and inspiring community, for teaching me so much about broader systems research, and for the random fun chats: George Amvrosiadis, David Andersen, Joy Arulraj, Nathan Beckmann, Lei Cao, Andrew Chung, Kevin Hsieh, Angela Jiang, Gauri Joshi, Anuj Kalia, Rajat Kateja, Saurabh Kadekodi, Christopher Canel, Jack Kosaian, Michael Kuchnik, Tian Li, Yixin Luo, Lin Ma, Prashanth Menon, Andy Pavlo, Gennady Pekhimenko (and for inviting me to his group meetings in UoF'T), Kai Ren, Majd Sakr, Alexey Tumanov, Dana Van Aken, Nandita Vijaykumar, Rashmi Vinayak, Daniel Wong, Lin Xiao, Huanchen Zhang, Qing Zheng, Giulio Zhou, and Timothy Zhu.

I want to thank the members and companies of the PDL Consortium including Alibaba, Amazon, Datrium, Facebook, Google, Hewlett Packard Enterprise, Hitachi, IBM Research, Intel Corporation, Micron, Microsoft Research, NetApp,
Oracle Corporation, Salesforce, Samsung Semiconductor Inc., Seagate Technology, and Two Sigma for their interest, insights, feedback, and support.

Sailing Lab was my major source of machine learning education. I thank Sailing Lab members whose time overlapped with mine, especially students who were involved in the Petuum project: Abhimanu Kumar, Seunghak Lee, Zhiting Hu, Pengtao Xie, Hao Zhang, and Xun Zheng, and Willie Neiswanger for doing a great job coordinating the group meetings.

My graduate study was complemented by two great summer internships with Microsoft Research and HP Labs. I thank my mentors and collaborators at MSR and HP Labs for helping me grow as a researcher: Madan Musuvathi, Todd Mytkowicz, Saeed Maleki, Adit Madan, Alvin AuYoung, Lucy Cherkasova, and Kimberly Keeton. I especially thank Madan for later hosting my job interview at MSR and giving me many career advices.

Qing Zheng became a close friend shortly after he joined CMU. I enjoyed our weekly dinners, where Qing many times generously fulfilled my curiosity of new research directions in storage systems. I also thank Qing for helping me improve my slides and give better presentations. Shen Chen Xu was my officemate for five years, and I enjoyed his company to dinners and numerous other events. Junchen Jiang and Kai Ren were among the first graduate students I met at CMU. I thank them for their advices as senior Ph.D. students. I thank all of my friends at CMU, who make graduate school much less stressful and much more fun.

I thank the staff of the Computer Science Department at CMU, especially Debbie Cavlovich, for taking care of me and making my life at CMU much easier.

I thank Sanjay Rao, Xin Sun, Vijay Raghunathan, and Carl Wassgren who mentored my undergraduate research at Purdue University and helped me with my graduate school application.

I am grateful to have Fan Yang standing by my side during some of the most stressful times. Thank you for your kindness, for cheering me up, for putting up with my childish acts, and for laughing at my stupid jokes.

Most importantly, I thank my parents, Xindong Wei and Xiuli Yi, and my grandparents, Chengpei Wei, Yanlan Liang, Hanxiu Yi, and Guiqin Ma for your never-ending love and support. I learned from you to be true to myself and to work hard and never give up. I would not have the freedom to chase my dream without the life that you provided me. Thank you for everything.
# Contents

1 **Introduction**
   1.1 Characteristics of Machine Learning Training Computation .............. 3
   1.2 Thesis Overview ......................................................................... 4
      1.2.1 Thesis Statement ............................................................... 4
      1.2.2 Contributions .................................................................... 5

2 **Background Concepts, Related Work and Trends** 7
   2.1 Distributed Computing Systems ............................................... 7
   2.2 Preliminaries on Machine Learning Training ............................... 9
   2.3 Strategies for Distributed Machine Learning Training ................. 9
      2.3.1 Data Parallelism ............................................................... 9
      2.3.2 Model Parallelism ............................................................ 11
   2.4 Related Work ....................................................................... 11
      2.4.1 Machine Learning Training Systems .................................. 11
      2.4.2 Communication Optimizations for Data-Parallel Training ...... 14
      2.4.3 Memory Optimizations for Deep Learning .......................... 15
   2.5 Machine Learning Trend: Increasing Model Computation Cost ...... 17
      2.5.1 More Complex Models ..................................................... 18
      2.5.2 Model Selection ............................................................... 20
   2.6 Machine Learning Systems Trend: From I/O to Computation .......... 20
      2.6.1 Deep Learning Compilers ................................................ 21
      2.6.2 Model Parallelism and Device Placement ............................ 22
3 Scheduling Inter-Machine Network Communication

3.1 The Bösen Parameter Server Architecture

3.1.1 System Architecture

3.2 Managed Communication

3.2.1 Bandwidth-Driven Communication

3.2.2 Update Prioritization

3.2.3 Adaptive Step Size Tuning

3.3 Evaluation

3.3.1 Communication Management

3.3.2 Comparison with Clock Tick Size Tuning

3.4 Summary

4 Application-Specific Computation Scheduling Case Study

4.1 LightLDA: Scheduling Computation for Latent Dirichlet Allocation

4.1.1 Introduction

4.1.2 Background: Latent Dirichlet Allocation and Gibbs Sampling

4.1.3 Scheduling Computation

4.1.4 Evaluation

4.2 Distributing SGD Matrix Factorization using Apache Spark

4.2.1 Introduction

4.2.2 Background: Spark and SGD Matrix Factorization

4.2.3 Communicating Model Parameters

4.2.4 Evaluation and Results

4.2.5 Discussion

4.3 Summary

5 Scheduling Computation via Automatic Parallelization

5.1 Dependence-aware Parallelization

5.2 Orion Programming Model
List of Figures

1.1 Cartoon depicting a typical training process: the model quality, as measured by the training objective function, improves over many update steps. The training algorithm converges when the model quality stops improving. 3

2.1 The hardware configuration of a node in a distributed cluster deploy at CMU (2016). 8

2.2 Comparing DRAM and GPU price 8

2.3 The computation cost to train stat-of-the-art models in Computer Vision and Natural Language Processing (source: Amodei et al. [10]). 18

2.4 ImageNet competition winners and runner-ups in recent years (source: [3]). 18

2.5 DNN Top-1 and Top-5 accuracy vs. computational complexity. Each ball represents a different DNN, and the size of the ball is proportional to the number of model parameters (source: [25]). 19

3.1 Parameter Server Architecture 25

3.2 Exemplar execution under bounded staleness (without communication management). The system consists of 5 workers, with staleness threshold $S = 3$. Worker 2 is currently running in clock 4, and thus, according to bounded staleness, it is guaranteed to observe all updates generated in the $4 - 3 - 1 = 0$-th clock tick (black). It may also observe local updates (green) as updates can be optionally applied to local parameter cache. Updates that are generated in completed clocks by other workers (blue) are highly likely visible as they are propagated at the end of each clock. Updates generated in incomplete clocks (white) are not visible as they are not yet communicated. Such updates could be made visible under managed communication depending on the bandwidth budget. 26

3.3 Compare Bösen’s SGD MF w/ and w/o adaptive revision with GraphLab SGD MF. Eta denotes the initial step size. Multiplicative decay (MultiDecay) used its optimal initial step size. 32
3.4 Algorithm performance under managed communication ........................ 36
3.5 Model Parameter Communication Frequency CDF ............................ 37
3.6 Overhead of communication management: time per data pass and average
 bandwidth consumption. Note that while managed communication co-
sumes high network bandwidth and takes longer to perform a mini-batch,
it significantly reduces the number of epochs needed to reach the target ob-
jective function value (see Fig. 3.4) and thus improves the wall clock time
to convergence (see Fig. 3.7) ......................................................... 38
3.7 Absolute convergence rate under managed communication .................. 39
3.8 Compare Bösen LDA with Yahoo!LDA on NYTimes Data ....................... 40
3.9 Comparing Bosen with simply tuning clock tick size: convergence per epoch 40
3.10 Comparing Bosen with simply tuning clock tick size .......................... 41
4.1 Partition the corpus dataset along by documents (horizontal) and words
(vertical); schedule a selected subset of partitions to run in parallel in each
step. An entire data pass is completed in a number of sequential steps. . . . 46
4.2 LightLDA log-likelihood over time. .................................................. 47
4.3 LightLDA breakdown of per-iteration time. ......................................... 47
4.4 Single-threaded baseline ............................................................... 54
4.5 Spark running on a single machine .................................................... 54
4.6 Strong scaling with respect to number of cores ................................... 55
4.7 Strong scaling with respect to number of machines ............................. 56
4.8 Weak scaling and cache misses ....................................................... 57
5.1 Data parallelism vs. dependence-aware parallelism: (a) the read-write (R/W)
sets of data mini-batches \( D_1 \) to \( D_4 \); (b) in data parallelism, mini-batches
are randomly assigned to workers, leading to conflicting parameter ac-
cesses; (c) in dependence-aware parallelization (note that \( D_4 \) instead of \( D_2 \)
is scheduled to run in parallel with \( D_1 \)), mini-batches are carefully sched-
uled to avoid conflicting parameter accesses. .................................... 60
5.2 Orion System Overview ............................................................... 61
5.3 Distributed parallel for-loop example ................................................. 63
5.4 SGD Matrix Factorization Parallelized using Orion .............................. 66
5.5 Overview of Orion’s static parallelization process using SGD MF as an example. .......................................................... 67
5.6 Overview of Orion’s static parallelization process using SGD MF as an example. .......................................................... 68
5.7 1D parallelization. ........................................................................................................................................ 71
5.8 1D computation schedule. ................................................................................................................................. 71
5.9 2D parallelization. ........................................................................................................................................ 72
5.10 2D computation schedule. ................................................................................................................................. 72
5.11 Unordered 2D parallel. ........................................................................................................................................ 72
5.12 Unordered 2D computation sched. ............................................................................................................................ 72
5.13 Pipelined computation of a 2D parallelized unordered loop on 4 workers . ....................................................... 74
5.14 Time (seconds) per iteration .............................................................. ........................................................................... 80
5.15 Orion parallelization effectiveness: comparing the time per iteration (averaged over iteration 2 to 8) of serial Julia programs with Orion-parallelized programs. The Orion-parallelized programs are executed using different number of workers (virtual cores) on up to 12 machines, with up to 32 workers per machine. ................................................................. 80
5.16 Orion parallelization effectiveness: comparing the per-iteration convergence rate of different parallelization schemes and serial execution; the parallel programs are executed on 12 machines (384 workers). ...................... 81
5.17 Bandwidth usage, LDA on NYTimes .............................................................. ........................................................................... 82
5.18 Orion vs. Bösen, convergence on 12 machines (384 workers) ......................... 83
5.19 Orion vs. STRADS, convergence on 12 machines (384 workers) ......................... 84
5.20 Orion vs. TensorFlow, SGD MF on Netflix .............................................................. ........................................................................... 85
6.1 TensorFlow Execution. Pattern indicates whether a node is a stateful (Variable or Constant) or stateless operation. Color indicates placement of the operation (CPU vs. GPU). ................................................................. 91
6.2 Mixture of Experts layer: example non-linear architecture. ................................................................. 93
6.3 Partition the computation graph to constrain memory consumption. Node color denotes expert partition. ................................................................. 95
6.4 Understanding TensorFlow Memory Consumption: Transformer w/ MoE ................................................................. 96
6.5 Placement optimization. ........................................................................................................................................ 98
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.6</td>
<td>Comparing graph partitioning strategies: DFS vs. Depth (depth-guided traversal) vs. DFS-Depth (DFS w/ depth-based prioritization)</td>
<td>99</td>
</tr>
<tr>
<td>6.7</td>
<td>The effect of graph partition size</td>
<td>100</td>
</tr>
<tr>
<td>6.8</td>
<td>Ablation study on a single GPU. Vanilla represents vanilla TensorFlow; +Partition represents TensorFlow with partitioned execution and memory swapping; +Placement represents placement optimization on top of +Partition.</td>
<td>102</td>
</tr>
<tr>
<td>6.9</td>
<td>VMesh-TensorFlow example. There are 6 physical devices arranged in a logical grid with cluster shape (3, 2). Each device is further partitioned with a device shape of (2, 2). The overall mesh used for compiling the Mesh-TensorFlow graph has shape (6, 4).</td>
<td>109</td>
</tr>
</tbody>
</table>
## List of Tables

2.1 Scaling model capacity in different ways. Results are collected from existing literature as cited. CV - Computer Vision, NLP - Natural Language Processing. ................................................................. 19

3.1 Bösen Client API ......................................................... 24

3.2 Datasets used in evaluation. Data size refers to the input data size. Workload refers to the total number of data samples in the input data set. ....... 33

3.3 Descriptions of ML models and evaluation datasets. The overall model size is thus # Rows multiplied by row size. .................................................. 33

3.4 Bösen system and application configurations. N - cluster Nome, S - cluster Susitna. The queue size (in number of rows) upper bounds the send size to control burstiness; the first number denotes that for client and the second for server. LDA experiments used hyper-parameters $\alpha = \beta = 0.1$. SGD MF and MLR uses an initial learning rate of 0.08 and 1 respectively. ....... 33

3.5 Summary of experiment result figures. ........................................... 34

4.1 Datasets used for the experiments. ............................................. 54

5.1 Comparing different systems for offline machine learning training. ....... 75

5.2 ML applications parallelized by Orion. ......................................... 78

5.3 Time per iteration (seconds) with ordered and unordered 2D parallelization (12 machines), averaged over iteration 2 to 100. ......................... 79

6.1 Deep Learning models (implemented on TensorFlow) used in our evaluation and the number of model parameters. ............................... 90
6.2 Graph statistics for the DNN models used in benchmarks. Depth refers to the length of the longest path. The number of parameters in MoE is tunable and we report the smallest version that we used in our benchmarks here. ................................................................. 90

6.3 Details of the benchmark implementations ........................................... 101

6.4 Average memory consumption and runtime overhead across all models. .... 103

6.5 The maximum supported mini-batch size by both systems ..................... 103

6.6 Throughput using the maximum supported mini-batch size. ...................... 104

6.7 Maximum ResNet model size that can be trained on a single Titan X GPU and computation throughput with different mini-batch size. ................. 105

6.8 Maximum number of experts that can be trained on a single TitanX GPU. We use a batch size of 8 and graph partition size of 200. ......................... 105

6.9 RNN training: time per mini-batch (seconds) for different input sequence length. ................................................................. 105

6.10 Maximum number of experts that can be trained on 4 nodes each with a single TitanX GPU. We use a batch size of 8 and graph partition size of 200. 106

6.11 Largest model configuration supported by Grappler Memory Optimizer and TensorFlowMem. ...................................................... 106

6.12 Grappler memory optimizer: simulator prediction and effectiveness. ...... 106

6.13 Maximum number of experts that can be trained on a single TitanX GPU. We use a batch size of 8 and graph partition size of 200. For VMesh-TensorFlow, we split the batch and experts dimensions of all tensors across a virtual mesh of size 4. ......................................................... 110

6.14 Maximum number of experts that can be trained on 4 nodes each with a single TitanX GPU. We use a batch size of 8 and graph partition size of 200. For VMesh-TensorFlow and SparseMoE, we split the experts dimension of all tensors across a virtual mesh of size 20 (cluster shape of 4 and device shape of 5). ......................................................... 111

7.1 Summary of memory optimization techniques and their trade-offs[81] ...... 113
Chapter 1

Introduction

In the early 2000s, the Google File System (GFS) [61] and the MapReduce system [51] showed that it is possible to store and process hundreds of TBs of data using thousands of machines that are composed on commodity hardware. Built on top of GFS, BigTable [29] supports efficient storage and retrieval of semi-structured data. Inspired by these systems, many open-source systems such as Hadoop (including HDFS) [4], HBase [5], and Spark [13], made this cost-effective solution available for the whole Internet industry. The availability of big datasets and distributed computing enabled machine learning techniques to be applied at increasingly larger scales, supporting more powerful applications. Compared to other data center applications, machine learning applications feature heavy and diverse mathematical computation, iterative processing, frequent and large volumes of network communication, and tolerance to bounded error. These distinctive characteristics present unique challenges and opportunities that call for new software systems. Below we briefly discuss some example applications of large-scale machine learning to motivate the need for machine-learning-specific software systems.

Ad click prediction. Online advertising typically relies on ad click prediction to serve ads to a proper audience to maximize profit. A natural approach to predicting the probability that an ad will be clicked if it is shown is logistic regression [111]. The logistic regression model, which is parameterized by a weight vector \( w \) of up to billions of dimensions and a bias \( b \), takes a feature vector \( x \) as input and outputs a probability (Eq. 1.1).

\[
p(x_i) = \sigma(w \cdot x_i + b)
\]  

The logistic regression model can be learned by optimizing a cross-entropy loss (Eq. 1.2, where \( y \) is the binary label) using different optimization algorithms, such as stochastic gradient descent (SGD) and coordinate descent. SGD repeatedly computes the model’s predictions for a small subset of observations (called a mini-batch) and updates the model parameters based on the difference between predictions and actual labels until the loss value stops improving, i.e., convergence.
\[
\arg \min_{w,b} L(w, b) = \arg \min_{w,b} \sum_{i=1}^{n} y_i \cdot \log(p(x_i)) + (1 - y_i) \cdot \log(1 - p(x_i))
\] (1.2)

**Recommender systems.** Due to the enormous size of their inventory, online shopping, and video streaming services, such as Amazon and Netflix, require personalized recommendation to help their customers find relevant merchandise or interesting videos [63]. Recommendation systems are popularly built using a matrix factorization model. Given a large (and sparse) \(m \times n\) matrix \(V\) (e.g., the user-item rating matrix in recommender systems) and a small rank \(r\), the goal of MF is to find an \(m \times r\) matrix \(W\) and an \(r \times n\) matrix \(H\) such that \(V \approx WH\), where the quality of approximation is defined by an application-dependent loss function \(L\). The \(W\) and \(H\) matrices can be solved by optimizing a nonzero squared loss (Eq. 1.3)

\[
\arg \min_{W,H} L_{NZSL} = \arg \min_{W,H} \sum_{i,j; V_{ij} \neq 0} (V_{ij} - [WH]_{ij})^2
\] (1.3)

**Topic modeling.** Topic modeling discovers the hidden topics of documents and is popularly used in online advertising, search engines, and recommendation systems. Latent Dirichlet Allocation (LDA) [26] has become the most popular model for topic modeling. The core of LDA is a topic distribution for each document and a word distribution for each topic. The most commonly used learning algorithm for LDA is collapsed Gibbs sampling, which learns the two distributions to maximize the likelihood of a model given the collection of documents. The collapsed Gibbs sampling algorithm sequentially samples a new topic for each word based on the current distributions and updates the distributions accordingly until the likelihood of the model stops improving.

**Image classification.** Deep learning has quickly become the most popular class of machine learning models in the past few years. The first widely successful application of deep learning is image classification [72, 94]. Given an input image, an image classifier outputs a label for that image, e.g., whether the image shows a cat or not. Today, image classifiers are commonly built using convolutional neural networks, which consist of many computation layers, and each with its parameters. The parameters in a convolutional neural network are commonly learned using stochastic gradient descent by minimizing a loss function that reflects the inaccuracy of the classifier.

**Other applications of deep learning.** Besides image classification, deep learning has been applied to improve the performance of existing machine learning applications and solve new problems. For example, many ad click prediction systems [70] and recommendation systems today are enhanced by deep learning [37, 43]. Deep learning achieved real-world success in many other applications, including, just to name a few, large-scale video analytics [79, 87, 166], machine translation [162], automatic email composition [30] and autonomous driving [53].
1.1 Characteristics of Machine Learning Training Computation

The above examples show many different models and learning algorithms, but they also share some common characteristics. More importantly, these characteristics are generally shared by typical machine learning training programs and can be leveraged to improve the execution efficiency of those programs.

![Diagram of training process]

**Figure 1.1:** Cartoon depicting a typical training process: the model quality, as measured by the training objective function, improves over many update steps. The training algorithm converges when the model quality stops improving.

**Iterative-convergent search for model parameter values.** Most commonly used training algorithms today are iterative-convergent. These algorithms search for model parameter values to optimize a certain objective function by refining the model parameters in small steps until convergence (Fig. 1.1). Due to the iterative-convergent nature of machine learning training, a training algorithm may produce many equally acceptable solutions – a set of model parameter values is considered an acceptable solution as long as the model quality is above a certain threshold.

**Large volumes of frequent parameter value updates.** Machine learning training algorithms are often inherently sequential, where a new model update is computed after the previous update is applied. Each update is often computed using a single data sample or a small mini-batch of data samples, so the training algorithm performs many update steps per pass over the training dataset. The training algorithm usually needs many data passes to produce a model that’s good enough. The high frequency of model parameter updates makes traditional batch processing frameworks, such as Spark [174], an inefficient option for distributed ML training due to its immutable data abstraction.

**Bounded-error tolerance.** Since the training algorithm is an iterative search process, intuitively, faulty update steps can be compensated for by taking some additional steps as long as the error is properly bounded. One important benefit of bounded error tolerance is that it offers a trade-off between computation throughput (number of data samples or updates per second) and computation quality. For example, previous work [75, 127] improve the
computation throughput of parallel and distributed training by violating the sequential semantics of the training algorithm.

**Increasingly high model complexity.** Machine learning is a fast-advancing field. The growing data size encourages researchers and practitioners to design increasingly complex models to improve performance and to support new applications. It is observed that across different application domains, more complex models often lead to better prediction accuracy. For example, over the past several years, winners of the ImageNet image classification competition have increasingly deep layering, achieving higher accuracy than the previous year’s winner. Besides depth, the increasing model complexity could also be due to larger layers (e.g., the Mixture of Experts [135]) and new computation-heavy operations, such as Capsule[74]. Due to the increasing model complexity, each update step performs more and more complex computation, and the training process requires more and more memory to store model parameters and intermediate results. The complex model computation demands more sophisticated optimizations and machine learning frameworks that employ fine-grained and informative intermediate representations of computation, such as dataflow graphs, to enable these optimizations.

1.2 Thesis Overview

1.2.1 Thesis Statement

**Thesis statement.** This thesis describes a set of system techniques that leverage the unique characteristics of machine learning training to improve computation efficiency. Collectively, these techniques support the following thesis statement:

*Machine learning training may leverage domain-specific opportunities to schedule network bandwidth, computation, and memory and achieve up to $5 \times$ faster training time and enable training up to $7 \times$ larger models.*

**Performance metrics of machine learning training.** The performance of most data center applications, such as batch processing systems, is usually quantified by computation throughput, which measures the amount of data processed or the number of queries served per second. Due to the iterative-convergent nature of machine learning training, the time to find an acceptable model, i.e., time to convergence depends both on the number of update steps per second, i.e., computation throughput, and the quality of each update step, i.e., convergence per data sample. Many of our techniques involve trade-offs between compute throughput and computation quality; therefore, we evaluate the performance of the machine learning training systems by time to convergence. We evaluate performance by computation throughput when such trade-offs are not involved.

**Programmable training systems.** The described techniques are implemented in Bösen and Orion, which are two distributed training systems that I developed from scratch, and TensorFlow, which is a widely popular deep learning system. These systems support a flexible
programming interface for application programmers to implement a wide range of machine learning models and algorithms. Our techniques introduce minimal, or even no, extra burden to application programmers and users. By using automatic parallelization, Orion substantially reduces programmer effort for distributed training compared to previous systems.

1.2.2 Contributions

I support the above thesis statement with three major research components.

Scheduling network bandwidth. Distributed machine learning training is often bottlenecked by limited network bandwidth. I design a communication management mechanism to better utilize network bandwidth that improves the convergence speed of distributed training. Its key idea is to selectively communicate a subset of messages based on their value when spare network bandwidth is available. This mechanism is implemented in Bösen, which is a Parameter Server system for data-parallel training. Experiments show that it outperforms the previous state-of-the-art synchronization mechanism by up to $5\times$. This research makes the following contributions:

- It introduces Bösen, which is one of the first general-purpose Parameter Server systems for data-parallel training.
- It describes a communication scheduling mechanism to improve inter-machine network communication efficiency in data-parallel training to improve convergence time.
- It presents experimental results on a wide range of machine learning models and algorithms to demonstrate the effectiveness of communication scheduling.
- As one of the earliest open-source machine learning systems, Bösen provides a testbed for future research on machine learning systems, such as LightLDA [171] and Poseidon [175].

Scheduling computation. Some model computation sparsely accesses model parameters when processing each data sample. Such sparsity may enable parallelization of the training algorithm that preserves its sequential semantics. However, leveraging this opportunity requires substantial programmer effort to analyze computation dependencies and parallelize the training computation manually. I design Orion, which is a new programming framework to automatically parallelize serial, imperative machine learning programs for distributed training. When applicable, Orion-parallelized ML programs converge faster than manual data-parallelism (even with communication scheduling) due to preserving the sequential semantics. Moreover, Orion falls back to data parallelism when permitted by the programmer to parallelize ML programs that are otherwise not sufficiently parallelizable. This research makes the following contributions:

- It introduces a holistic approach for automatically parallelizing serial ML programs for distributed computation, which includes data abstraction, programming model,
and auto-parallelization algorithm. Through this approach, a serial, imperative ML program can be parallelized with minimal changes. The auto-parallelization algorithm supports semantic relaxations tailored for parallelizing ML programs, which can be enabled by programmer hints.

- It describes the system Orion, which is an implementation of the above approach, which parallelizes ML application programs implemented in a scripting language (Julia [24]). Orion also features a new programming abstraction that unifies dependence-aware parallelization and data parallelism and supports a wide range of ML applications.

- It presents a comprehensive experimental evaluation of Orion that compares Orion with a number of existing ML systems and demonstrates the effectiveness of Orion’s parallelization.

Scheduling memory. As ML models become more and more complex, ML training demands higher and higher memory capacity to store model parameters and intermediate states. However, GPUs, which are the most widely used deep learning accelerators today, have limited memory, and are highly expensive. I design a memory scheduling mechanism that leverages the cheap host memory to store model parameters and intermediate results, which are prefetched to GPU memory when needed. In contrast to classic paging techniques, we leverage the computation graph to schedule data movement before the data is needed to avoid stalling GPU computation. Compared to vanilla TensorFlow, our technique enables training models with $4.4 \times$ more parameters on a single GPU and models with $7.5 \times$ more parameters on 4 distributed GPUs. This research makes the following contributions:

- It presents a model-agnostic approach to reduce GPU memory consumption during training by leveraging the cheap host memory. Our approach leverages the general dataflow graph to reduce the overhead of additional data movements.

- It describes an implementation of our techniques in TensorFlow, which is the most popular and most sophisticated deep learning system today. Our implementation does not introduce new programming interfaces and supports existing TensorFlow applications without modifications.

- Unlike previous works that are primarily evaluated on convolutional neural networks, we present a comprehensive evaluation across a wide range of deep learning models and successfully demonstrate the effectiveness of our approach.
Chapter 2

Background Concepts, Related Work and Trends

2.1 Distributed Computing Systems

Distributed computing clusters composed of commodity hardware are widely used for data-intensive applications, which are both deployed in private data centers and offered as public cloud services, e.g., Amazon AWS, Microsoft Azure, and Google GCP. The success of distributed computing owes much to sophisticated software systems that make it easy for application programmers to leverage the power of the large amount of inexpensive and unreliable hardware. These software systems include infrastructures that provide resource sharing among applications and data storage, as well as programming frameworks that target different application domains, such as batch processing, stream processing, and ML training. Machine learning systems often interact with other software systems running in the cluster.

Traditionally, distributed computing clusters consist of hundreds to thousands of CPU servers connected by 1 and 10 Gbps Ethernet. Increasingly more hardware accelerators, such as GPUs and TPUs and new interconnect technologies, such as NVLink, 100 Gbps Ethernet, RDMA, and Infiniband, are deployed to meet the growing needs of applications in recent years.

Bandwidth bottlenecks. Fig. 2.1 shows the hardware configuration of a node in a distributed cluster deployed at CMU in 2016, which represents the typical characteristics of cluster servers. Hardware characteristics reveal potential bottlenecks of the software systems running on top of it. First of all, the inter-machine network bandwidth is highly limited. Many older clusters employ Ethernet of 1 Gbit/s, and the network bandwidth on public clouds is commonly below 10 Gbit/s. Thus it is critical for a distributed system to avoid extensive network communication and make use of this scarce resource carefully. While GPU provides high compute power, the bandwidth between GPU and main memory is limited; and the I/O bandwidth between main memory and external storage such as hard drive disks and SSDs is even lower. In earlier systems, CPU cores share one system bus to access memory and experience the same bandwidth and latency when accessing different
memory regions, referred to as Uniform Memory Access (UMA). As the number of CPU cores increases, the per-core bandwidth of a UMA system scales poorly due to the limited scalability of the shared bus. As a solution, Non-Uniform Memory Access (NUMA) systems emerge, in which each processor has its local memory module (or zone) \[^9\]. A processor accesses remote memory modules via point-to-point connections between processors, such as the QPI bus on Intel processors, and experience considerably higher latency and lower bandwidth (e.g., up to 19.2 GB/s via a QPI bus).

**Figure 2.2:** Comparing DRAM and GPU price. GPU price is presented as dollars per MB of on-board memory. DRAM price was collected by John C. McCallum. \(^1\)

**Limited and expensive GPU memory.** While originally designed for computer graphics, GPUs are widely used today to accelerate deep learning due to its massively parallel cores and high memory bandwidth. Due to technological limitations, GPU memory cannot pro-
vide high bandwidth and high capacity at the same time. GPUs that are most commonly used for deep learning training today are limited to 12 or 16 GB of memory, and they are expensive. Fig. 2.2 compares DRAM price with the price of several desktop and server GPUs that are popularly used for neural network training, in terms of $ per MBytes of on-board memory. We observe that GPU price is not affected by the decreasing DRAM price and remains highly expensive. While the recently released Nvidia Tesla V100 GPU has 32 GB of memory\(^2\), it’s $1500 more expensive than the 16GB version without additional compute power, leading to a high $0.085 per extra MByte, higher cost than the entire price of the most cost-effective gaming card.

### 2.2 Preliminaries on Machine Learning Training

Training a model is essentially finding a set of model parameter values that optimize a certain objective function. This is typically done using an iterative convergent learning algorithm, which can be described by Alg. 1.

```
Algorithm 1: Serial Execution

\[
t \leftarrow 0
\]
\[
\text{for } \text{epoch} = 1, \ldots, T \text{ do}
\]
\[
\text{for } i = 1, \ldots, N \text{ do}
\]
\[
A_t + 1 \leftarrow A_t \oplus \Delta (A_t, D_i)
\]
\[
t \leftarrow t + 1
\]
```

In Alg. 1, \(A_t\) denotes the parameter values at time step \(t\), and \(D_i\) denotes the \(i\)-th mini-batch in the training dataset \(D = \{D_i|1 \leq i \leq N\}\). \(D_i\) may contain one or multiple data items. The update function \(\Delta()\) computes the model updates from a mini-batch of data items and the current parameter values, which are applied to generate a new set of parameter values. \(\Delta\) may include some tunable hyperparameters, such as step size in a gradient descent algorithm, which require manual or automatic tuning for the algorithm to work well. \(\oplus\) represents the operation to apply parameter updates, which is usually addition. The algorithm repeats many times (i.e., epochs) until \(A\) stops changing, i.e., converges. In each epoch, the algorithm takes a full pass over the training dataset.

### 2.3 Strategies for Distributed Machine Learning Training

#### 2.3.1 Data Parallelism

The most commonly used approach for parallelizing machine learning training is data parallelism. Data parallelism parallelizes the inner for-loop that iterates over mini-batches \(D_i\) by processing many (and even all) mini-batches in parallel with respect to each worker’s local model state. Note different mini-batches may read and write the same set of model

\(^2\)Price according to thinkmate.com
parameters, and in serial execution, a later mini-batch observes the updates generated by the previous mini-batches. However, in data parallelism, mini-batches do not observe updates from other parallel mini-batches until their updates are propagated to a worker’s local model state.

**Bulk synchronous parallel.** Bulk synchronous parallel (BSP) [147] is a commonly used synchronization mechanism for data-parallel training. Under BSP, workers alternate between computation and synchronization. After computing a local computation step, each worker enters a synchronization phase. During the synchronization phase, each worker propagates update messages generated from its local computation to other workers and receives others’ updates. Thus the synchronization phase does not end until all workers finish communicating updates. Such a parallelization model is simple but is prone to poor performance since all workers proceed at the speed of the slowest worker. The BSP model does not necessarily preserve a serial algorithm’s sequential semantics unless each worker works on independent computation within each iteration.

Due to the iterative nature of machine learning training, in existing literature, the term iteration has been overloaded to refer to a full data pass over the training dataset, i.e., an epoch or a mini-batch, depending on the context. To avoid any confusion, throughout this thesis, we refer to a full pass over the training dataset as one epoch and use iteration to refer to the repeated step in an iterative execution model, such as BSP.

**Local buffering.** When the model computation is light, computing updates from a single mini-batch takes little time compared communicating model updates over the bandwidth-limited inter-machine network. Thus communicating once per update step incurs considerable communication overhead. Since the model updates are usually additive, the communication overhead can be reduced by locally buffering the updates and communicating updates once per $N$ update steps, which allows coalescing delta changes to reduce the total communication volume. However, local buffering incurs a higher staleness in parameter states. Larger staleness causes larger inconsistency compared to serial execution and may slow down the per-data-sample convergence rate.

**Totally asynchronous parallel.** In order to overcome the communication overhead and mitigate waiting for stragglers, people also proposed totally asynchronous parallel (TAP) in contrast to BSP. In TAP, a worker propagates parameter updates and fetches new parameter values (typically from a set of servers referred to as Parameter Server) without waiting for other workers. Additionally, a worker proceeds to the next local computation step using locally cached stale parameter states without waiting for the new parameter values. TAP achieves high computation throughput. However, staleness may be arbitrarily large and even lead to divergence.

**Stale synchronous parallel (or bounded staleness).** Motivated by the staleness problems of TAP, stale synchronous parallel (SSP) ensures bounded staleness by blocking a worker’s computation when its locally cached parameter states are more than $T$ steps stale. This also
means the fastest worker cannot be more than $T$ steps ahead of the slowest worker. Previous work proves that convergence is guaranteed for certain models when step size is properly tuned [75].

### 2.3.2 Model Parallelism

Model parallelism broadly refers to parallelization strategies where different workers work on different parts of the model.

**Select data samples to process in parallel.** Some ML models exhibit a sparse access pattern where the update computation function $\Delta (A_t, D_i)$ reads and updates a small subset of the model parameters. By carefully choosing which mini-batches to run in parallel, the parallel workers work on disjoint subsets of model parameters. As demonstrated by Kim et al. [90], such a parallelization typically preserves the sequential semantics of the learning algorithm and thus achieve a higher per-data-sample convergence rate. However, it usually requires non-trivial programmer effort to manually analyze data dependence and implement an efficient distributed program.

**Distribute the computation of a single update.** For sufficiently complex models, such as deep neural networks, the update computation $\Delta (A_t, D_i)$ is large enough and thus worthwhile to be parallelized. The computation of $\Delta ()$ can be distributed among multiple, even heterogeneous devices. The effectiveness of such parallelization depends on the parallelism of the function $\Delta ()$. DistBelief [52] is an early example of partitioning the model computation across multiple machines. TensorFlow supports user-defined device placement specification at the granularity of individual operations. Mirhoseini et al. [115, 116] and Jia et al. [86] propose different approaches to automate device placement to achieve better training performance. Harlap et al. [71] also leverage pipeline parallelism across mini-batches to improve the utilization of parallel computing resources.

### 2.4 Related Work

#### 2.4.1 Machine Learning Training Systems

Over the last decade, many systems have been developed for large-scale machine learning training. These systems aggressively leverage the application-specific properties of the machine learning models and algorithms to improve system execution efficiency. Machine learning is a fast-advancing field. New models and algorithms are frequently invented, and the application domain of machine learning is also fast expanding. Advances in machine learning introduce new important workloads that pose new challenges and present new opportunities for systems. As a result, large-scale ML systems are fast evolving as well. In this chapter, we briefly discuss representative existing machine learning systems, which offer insights that can be leveraged by future systems.
Distributed Implementations of Machine Learning Applications

Some machine learning models, such as Latent Dirichlet Allocation for topic modeling, have found diverse and essential use cases in the industry. Such use cases motivate distributed implementations of such models to enable training on massive datasets. Such systems include Yahoo!LDA [18], Peakcock [155], XGBoost [32], and Caffe [85]. Since they target specific machine learning use cases, they often lack a programming interface and sometimes rely on configuration files to express variations in the model or learning algorithm.

Batch Processing Systems

General-purpose batch processing systems, such as Hadoop [4] and Spark [174], support a programming interface for distributed execution. Many attempts were made to implement large-scale machine learning training on these systems, including most notably MLLib [8] on Spark. However, these systems are not suitable for machine learning training as they lack an abstraction and efficient implementation for frequently mutated states. This limitation prevents batch processing systems from achieving high training speed and training large models.

Graph Processing Systems

The ubiquitous graph datasets draw the attention of data mining and machine learning researchers. An early attempt to design a programming interface for machine learning training is thus specialized in graph processing. Notable examples include GraphLab [105], PowerGraph [64], GraphChi [96]. These systems feature a vertex programming abstraction, where users implement a vertex program that executes on each vertex of the data graph. The vertex program has a well-defined access pattern, i.e., it may only access neighboring vertices and edges, which enables many opportunities for optimizations, such as partitioning the data graph to minimize cross-machine communication. While vertex programming is well suited for many graph mining applications, it is highly restrictive for other machine learning applications. As machine learning models become more and more complex, and the model states associated with each vertex and each edge becomes larger, the training application is more and more bottlenecked by model computation. While the vertex programming model enables optimizations for disk and network I/O, it is cumbersome for application programmers to implement computation optimizations as the vertex program has only a local view of the computation and states.

General-Purpose Parameter Server Systems

By adopting a distributed shared memory (DSM) abstraction, Parameter Server systems, such as LazyTable [44], IterStore [45], and parameter server [100] provide shared access to model parameters among distributed training programs. The low-level and primitive DSM interface offers great flexibility for machine learning applications but relies on sophisticated application implementations to achieve high computation throughput.
Deep Learning Systems

Deep neural networks (DNNs) have become one of the most popular classes of machine learning models in recent years. A DNN model usually consists of a sequence of cascaded functions that transform an input $x$ to some prediction $y$ (Eq. 2.1). Each function (commonly referred to as a layer) is typically parameterized by a few dense matrices, and the computation involves matrix multiplications and additions. The high complexity of matrix operations and the large number of layers makes it computationally expensive to evaluate DNNs.

\[
y = f_n \circ f_{n-1} \circ \ldots \circ f_1 (x)
\]  

(2.1)

Many frameworks have been developed for deep learning, including early efforts such as Caffe [85], DistBelief [52], and Project Adam [39]. Caffe and DistBelief represent the neural networks as a sequence of layers and perform fixed training computation over the neural network definition. This representation makes it difficult for machine learning researchers to define new layers and experiment with new or refined training algorithms. Motivated by this challenge, modern deep learning frameworks, such as TensorFlow [16] and MXNet [33], represent the neural network computation as a dataflow graph consisting of fine-grained primitive operations, which makes it simpler to define new layers and new training algorithms. The computation graph provides a global view of the training computation and thus enables many optimization opportunities, such as operator fusion, data layout transformation, and dead code elimination. Instead of relying on a computation graph as the intermediate representation, PyTorch [123] offers a more programmer-friendly imperative programming interface but misses the optimization opportunities that a computation graph would have enabled. As an extension to TensorFlow, TensorFlow Eager [?] supports imperative programming using TensorFlow operations and kernels to lower the burden of TensorFlow users. In order to achieve the high performance of TensorFlow graph execution, TensorFlow Eager introduces a Python decorator function, which traces a Python function to create a computation graph for just-in-time compilation. PyTorch’s JIT compiler (torch.jit.trace) similarly traces the imperative execution to build a computation graph in order to enable automatic differentiation [123]. The aforementioned tracing approach often fails to correctly capture dynamic features in an imperative Python program, such as dynamic control flow, dynamic data types, and impure functions. JANUS [84] speculatively executes the computation graph that’s constructed by tracing and falls back to imperative execution when the actual execution differs from the trace. AutoGraph [117] leverages static analysis of the Python code to correctly transform dynamic control flows.

Online Learning Systems

So far, we have focused on batch training systems, where the training data is collected and prepared before training, and a machine learning model is trained from scratch. However, in many applications, the relationship between input data and output labels can change
over time, which is referred to as concept drift [59]. When such changes happen too rapidly, it might be too slow or too expensive to re-train the model from scratch to adapt to the changing environment. Online learning is thus proposed to incrementally update a machine learning model based on new observations while it is served online. One notable example of online learning is the ad click prediction system deployed at Google [111].

2.4.2 Communication Optimizations for Data-Parallel Training

Overcoming the network communication bottleneck has been a focus of distributed machine learning training systems since the early days and is a focus of this thesis. In this section, we review the existing literature on communication optimizations for distributed machine learning training.

Graph Partitioning

In many graph processing applications, the access pattern is characterized by the data graph itself, i.e., processing a vertex reads and updates its neighboring vertices and edges. Partitioning the data graph while minimizing cut edges reduces inter-machine communication volume [106].

In sparse models such as sparse logistic regression, a subset of model parameters is read and updated when processing each data sample. Placing data samples that share access to many model parameters on the same machine and placing the accessed model parameters accordingly reduces inter-machine network communication volume. This access pattern can be characterized as a bipartite graph, and this problem can be solved by partitioning the graph with minimal edge cut while balancing the size of each partition [101].

Local Buffering

Early graph processing systems [64] and parameter server systems [44, 45, 75] usually aggressively buffer updates locally, e.g., synchronize once per epoch (data pass), to reduce the synchronization overhead. The locally buffered updates can be optionally applied to update the worker’s local model cache. The computation-to-communication ratio increases as the machine learning models become more complex, and faster interconnect technologies are deployed in data centers. Thus local buffering has become less popular for training complex DNNs. However, it remains an important optimization for machine learning under limited network bandwidth, such as in federated learning [112].

Our thesis proposes a mechanism to adaptively tune the network communication frequency based on available network bandwidth. Wang et al. [152] verified that the training algorithm, e.g., SGD, tolerates higher staleness in the beginning but becomes more and more sensitive to staleness as the algorithm converges. Therefore, they propose an algorithm to adaptively tune the synchronization frequency based on the learning progress.
Data Compression

Data compression has been applied to various types of data to reduce the storage cost and I/O overhead, including images (e.g., JPEG [151]), audio (e.g., MP3 [132]), and video (e.g., MPEG [38]). Machine learning training enjoys domain-specific opportunities for data compression to reduce the communication volume. Xie et al. [164] found that in some machine learning applications, large dense matrices can be factored into the outer product of two vectors. This lossless compression scheme requires each worker to broadcast its updates to all other workers and thus can be applied to reduce the communication volume on a small scale.

While Bösen exploits the magnitude of the delta updates to prioritize messages for communication when exploiting spare network bandwidth, other parameter server systems leverage low magnitude to suppress communication when the network bandwidth is highly limited. Hsieh et al. [78] communicate only updates that are significant enough over WLANs for efficient geographically distributed training. Aji et al. [20] and Lin et al. [104] propose to communicate only significant gradients when training DNNs in a data center, dropping or delaying insignificant gradients. Wen et al. [158] show that gradients can often be represented using only 3 bits while achieving reasonable model performance in distributed training.

Scheduling Communication Based on Access Pattern

In complex models such as DNNs, the model parameters are not all accessed at the same time. Parameter synchronization can be scheduled according to the order the parameters are accessed in the worker program to avoid blocking communication. Jayarajan et al. [82] demonstrated the effectiveness of this approach on MXNet and showed an up to 66% improvement in computation throughput without sacrificing convergence speed. Peng et al. [124] leverage this idea to build a generic communication scheduler that is applicable to TensorFlow, PyTorch, and MXNet.

2.4.3 Memory Optimizations for Deep Learning

The growing tension between increasing model complexity and limited and expensive GPU memory motivates researchers to develop memory optimization techniques to reduce memory consumption during training. In this section, we review prior works on memory optimizations for training neural networks.

The parameters of a DNN are typically learned using stochastic gradient descent (SGD), where the gradients are computed using back-propagation [133]. In Eq. 2.1, assuming the parameters of function \( f_i(x_{i-1}) \) are \( w_i \), SGD requires computing the gradients \( \frac{\partial y}{\partial w_i} \) for all functions \( f_i \). Backpropagation computes \( \frac{\partial y}{\partial w_i} \) using the chain rule (Eq. 2.2).

\[
\frac{\partial y}{\partial w_i} = \frac{\partial f_n}{\partial f_{n-1}} \cdot \cdots \cdot \frac{\partial f_i}{\partial w_i}
\]  

(2.2)
Back-propagation requires the intermediate results $f_{n-1}, \ldots, f_i$ to compute the gradient $\frac{\partial y}{\partial w_i}$. These intermediate results (often referred to as activation values) can be stored in memory to avoid recomputation. In TensorFlow, this is achieved by reference counting, i.e., the gradient computation operations hold a reference handle to the relevant intermediate results. Storing the intermediate results constitutes the major source of memory consumption for many neural networks [128] and becomes the main target for memory optimizations.

Gradient Checkpointing

Chen et al. [35] proposes to checkpoint only a subset of intermediate results in a sequence of functions, and recompute the rest when necessary to reduce memory consumption. The idea of trading off recomputation for memory has been investigated in the automatic differentiation community [67]. Specifically, Chen et al.’s algorithm achieves $O(\sqrt{N})$ memory consumption at the cost of one additional forward pass for a sequence of $N$ operations by partitioning the sequence into $\sqrt{N}$ segments, storing only the outputs of the endpoints, and recomputing each segment during the backward pass. Gruslys et al. [69] specifically focus on recurrent neural networks and designed a dynamic programming algorithm to maximize computation throughput under memory constraints. Salimans et al. [146] implement Chen et al.’s algorithm for TensorFlow applications.

Memory Swapping

During the back-propagation of a mini-batch, not all the parameters and activation values are needed at all the time. This observation inspires a series of works to reduce GPU memory consumption by offloading parameters and activation values to cheaper host memory and loading them only when they are needed. Cui et al. [46] is implemented for Caffe and uses the coarse-grained layer as the unit of swapping operations. Rhu et al. [128] recognize that the convolution layers are computationally heavy, and their outputs consume a large amount of memory, making them a good target for memory offloading. Thus besides offloading all layers, Rhu et al. propose another mechanism to offload only convolution layers to achieve high computation throughput with higher memory consumption. Wang et al. [153] notice some neural networks are not simply a linear sequence of layers, such as Inception [144], and thus linearizes the layers by traversing the neural network in a topologically sorted order. These techniques are applied to coarse-grained layer-wise neural network representations and are mostly evaluated on convolutional neural networks. Meng et al. [113] describes a memory swapping mechanism for the fine-grained operation-wise graphs in TensorFlow and TensorFlow’s Grappler memory optimizer implements a similar memory swapping mechanism [14]. These mechanisms rely on accurate estimations of operations’ execution time and memory usage and insert memory swapping operations to the graph when memory swapping does not slow down graph execution based on the simulation.
Mixed Precision and Quantization

Normally the weights, activation values, and gradients in a neural network are represented as 32-bit (single precision) floating-point numbers, i.e., FP32. There have been many previous works on using lower-precision representations for the neural network weights, activations, and gradients to reduce the computation and memory overhead both during training and during inference. In this section, we briefly review some most representative works.

Micikevicius et al. [114] store a master copy of the neural network weights in FP32 but uses a 16-bit floating-point (FP16) copy of the weights to compute activations and gradients, which are also stored in FP16. With the help of loss scaling, Micikevicius et al. show that a number of convolutional neural networks can be trained with mixed-precision floating-point numbers without loss of accuracy.

A number of works propose to represent the weights as fixed-point numbers, most commonly, 8-bit integers (INT8) and perform fixed-point arithmetic for training [42]. Some more aggressive quantization works represent weights and activations as binary [41] or ternary values [98].

Compression

Jain et al. recognizes the output of some important neural network operations such as ReLU layers followed by a pooling layer and ReLU followed by convolution layers can be losslessly compressed and achieve a high compression ratio [81]. Jain et al. also performs lossy compression on the activations used in the backward pass using low-precision representation.

2.5 Machine Learning Trend: Increasing Model Computation Cost

The focus of large-scale machine learning systems shifts as newer important machine learning models emerge. While traditionally machine learning systems have focused on graph processing applications and models such as sparse logistic regression, LDA and matrix factorization, which exhibit a sparse parameter access pattern and have low computational complexity for each mini-batch, machine learning systems today are focused on deep neural networks that exhibit a dense parameter access pattern and much higher per-mini-batch computational complexity. Advances in the field of machine learning drive the direction of machine learning system research.

Amodei et al. [10] at OpenAI calculated the amount of compute (in Petaflop/s-day) needed to train popular deep learning models that are proposed in the past 10 years, which is shown in Fig. 2.3. They argue that the amount of compute used in the largest machine learning training runs has been increasing exponentially with a 3.5 month-doubling time. This fast growth of the compute needed for training is a direct consequence of the increasing model complexity.
Figure 2.3: The computation cost to train state-of-the-art models in Computer Vision and Natural Language Processing (source: Amodei et al. [10]).

Figure 2.4: ImageNet competition winners and runner-ups in recent years (source: [3]).

2.5.1 More Complex Models

More complex models for better performance. It has been widely observed that across various computer vision and natural language processing applications, more complex models achieve higher accuracy. Fig. 2.5 shows some representative deep neural networks that were developed in recent years for image classification. Those DNNs lie on a curve starting from the lower-left corner and going up to the upper right corner, and the size of the ball, i.e., the number of model parameters increases along this direction. This trend indicates that larger models achieve higher accuracy and incur larger computation overhead. Fig. 2.1 shows that across different applications, for the same model architecture, increasing model capacity improves model accuracy. As a consequence, deep learning models are becoming more
and more complex over time (Fig. 2.1).

![Figure 2.5: DNN Top-1 and Top-5 accuracy vs. computational complexity. Each ball represents a different DNN, and the size of the ball is proportional to the number of model parameters (source: [25]).](image)

<table>
<thead>
<tr>
<th>Model Class</th>
<th>Application</th>
<th>Metric</th>
<th>Scale By</th>
<th>Scale</th>
<th>#Param.</th>
<th>Perf.</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG [139]</td>
<td>CV</td>
<td>top-1 error</td>
<td>layers</td>
<td>16 layers</td>
<td>138M</td>
<td>25.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>19 layers</td>
<td>144M</td>
<td>25.5</td>
</tr>
<tr>
<td>ResNet</td>
<td>CV</td>
<td>top-1 error</td>
<td>layers</td>
<td>50 layers</td>
<td>N/A</td>
<td>20.74</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>101 layers</td>
<td>N/A</td>
<td>19.87</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>152 layers</td>
<td>N/A</td>
<td>19.38</td>
</tr>
<tr>
<td>MoE [135]</td>
<td>NLP</td>
<td>test perplexity</td>
<td>experts</td>
<td>32 experts</td>
<td>100M</td>
<td>40.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4096 experts</td>
<td>4.4B</td>
<td>30.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>65536 experts</td>
<td>9.2B</td>
<td>28.9</td>
</tr>
<tr>
<td>ResNet-40</td>
<td>CV</td>
<td>test error</td>
<td>widening</td>
<td>1 × wide</td>
<td>0.6M</td>
<td>6.85</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4 × wide</td>
<td>8.9M</td>
<td>4.97</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8 × wide</td>
<td>35.7M</td>
<td>4.17</td>
</tr>
</tbody>
</table>

Table 2.1: Scaling model capacity in different ways. Results are collected from existing literature as cited. CV - Computer Vision, NLP - Natural Language Processing.

**New model architectures.** Machine learning researchers constantly design new models to improve model accuracy and reduce computation cost. New models use new operators, such as Capsule [74], as well as novel structures, such as residual connections [72], attention [149], and the Mixture-of-Experts layer [135]. Deep learning frameworks often need to be extended with new kernels, APIs, and optimizations passes in order to support these new models efficiently.
2.5.2 Model Selection

The computation cost of machine learning training is amplified by the search for optimal training hyperparameters and recently, search for neural network architectures. The search space is large due to the inter-dependence between different hyperparameters and network architectures. These search processes involve training multiple versions of the same model or similar models with different hyperparameters and select the version that achieves the best accuracy. The high computational cost motivates ML researchers to design new search algorithms and system optimizations to reduce the overhead of training many model versions. HiveMind [121] proposes to batch operations and share input data across multiple model executions in order to improve the efficiency of hyperparameter search and neural architecture search.

Hyperparameter search. Machine learning training algorithms, such as the widely used gradient descent algorithm, involve hyperparameters, such as step size. Due to the ambiguity of the effect of hyperparameter values on model accuracy, grid search with repeated trials become the most popular practice. Notable approaches to automate hyperparameter search and reduce its computation overhead include Spearmint [140], HyperBand [99], and MLtuner [47].

Neural architecture search. Common practices to designing deep learning models often involve tuning the neural network architecture, for example, changing the number of layers, using different layers, or changing layer sizes. In order to reduce accelerate and improve upon the time-consuming and error-prone model design process by human experts, there is increasing interest in machine-learning-automated neural architecture search. Generally speaking, the machine-learning-automated search trains a controller (often using reinforcement learning) that samples task networks (e.g., for image classification) with different architectures. The task networks are trained to convergence to obtain some accuracy, which is used as the reward to generate updates to the controller network. Each update to the controller involves training multiple task networks, typically in parallel, and each task network is usually trained using data parallelism. Thus training the controller network demands a large number of computing resources. For example, Zoph et al. (2017) [179] and Zoph et al. (2018) [180] respectively used 800 and 500 GPUs to train the task networks and took 28 days and 4 days to yield the desirable task networks. Recent research, such as ProxyLess-NAS [27], has been devoted to reducing the computation overhead of neural architecture search while achieving comparable or even better task model accuracy than previous work.

2.6 Machine Learning Systems Trend: From I/O to Computation

As the machine learning models become more and more complex and training demands more and more compute power, machine learning systems emphasize more and more on computation. Earlier graph processing systems such as GraphChi [96] and X-Stream [131] focus on reducing the I/O overhead of loading data from external storage as the target applications, such as PageRank, connected components, and triangle counting, traverse the
data graph but perform light computation on each vertex. Distributed graph processing systems, e.g., GraphLab [106] and PowerGraph [64], and Parameter Server systems focus on reducing network communication overhead. As the disk and network I/O overhead reduces and the computation overhead increases, more and more effort has been devoted to optimize model computation in deep learning systems, both for training and inference.

In this section, we discuss some promising recent directions in machine learning systems to overcome the computation overhead in DNN training and inference. In particular, these new directions feature machine-learning-guided search to improve upon heuristics designed by domain experts. This machine-learning-guided search measures the computation throughput of a wide range of optimization configurations, which is used to train a model that predicts the computation throughput of a given configuration or outputs the optimal configuration. It leverages the observation that the same computation graph is repeatedly applied to all data samples, and thus the computation throughput measured on a random input is representative of all data samples.

### 2.6.1 Deep Learning Compilers

Traditionally, deep learning frameworks rely on hand-optimized operation kernels to achieve high computation efficiency. The diverse set of operations and the complex interactions between them demand optimizations across operation boundaries, such as operator fusion and data layout transformation. Therefore, several projects, including TensorFlow XLA [15], propose to optimize DNN models by transforming the computation graph. However, heuristic-based optimizations are limited by scarce and expensive human time and thus naturally target standard benchmarks and widely-used models. New models often introduce new operations and expose fresh optimizations opportunities. Moreover, various custom deep learning accelerators are emerging for deployments in data centers as well as edge devices. Existing DNNs need to be optimized for new hardware, which exposes different architectures and benefits from different optimization heuristics. Hand-optimized implementations also involve numerous parameters, such as tile size, which need to be set differently for different hardwares. A number of academic and industrial projects try to leverage machine learning to automate optimizations with little to no human intervention. Notable examples include TVM [36], PlaidML [11], and Tensor Comprehensions [148]. Note that published results are more focused on inference with growing support for training.

Existing dataflow-graph-based representations of DNN models have a number of limitations compared to the intermediate representations of general-purpose programming languages. For example, dataflow graphs have limited support for dynamic control flow and do not support functions. Deep learning models suffer more and more from these limitations as the model computation becomes more complex. Recent works, including MLIR [97] and Relay [130], propose to use more general intermediate representations to represent DNN computation.
2.6.2 Model Parallelism and Device Placement

As DNN models become more and more complex, the model computation is more and more limited by the computing power and memory capacity of a single computing device. Distributing the model computation among multiple computing device provides a solution to this bottleneck in many deep learning applications.

TensorFlow [16] relies on application programmers to manually place operations on devices. GPipe [80] and PipeDream [71] propose to partition the neural network among distributed devices in a layer-wise fashion. This coarse-grained partitioning potentially misses optimization opportunities and may fail to scale extremely large layers, such as the Mixture-of-Experts layer [135]. Mesh-TensorFlow [136] partitions a large operation across distributed computing devices based on a user-provided mesh layout and thus enables training DNNs with operations that have large inputs or outputs. Mirhoseini et al. [115, 116] learn the device placement of individual operations from repeated trial executions of various schedules. Jia et al. [86] simulate the execution to reduce the planning cost down to sub-seconds to tens of minutes depending on the scale (4 to 64 GPUs) and the complexity of the network. Moreover, Jia et al. [86] exploit additional dimensions of parallelization, such as intra-operation parallelism.
Chapter 3

Scheduling Inter-Machine Network Communication

As discussed in Sec. 2.3.1, data parallelism is one of the most popular parallelization strategies for distributed training, and local buffering is commonly used to balance the parameter synchronization overhead and convergence rate. An aggressive buffering strategy delays parameter synchronization to achieve a high computation throughput but suffers from slower convergence due to higher staleness in parameter values. While more frequent synchronizations may improve convergence rate, the synchronization frequency is ultimately limited by the network bandwidth.

In this chapter, I present Bösen, a Parameter Server system that’s designed to better utilize the precious network bandwidth for distributed data-parallel training by incorporating knowledge of network bandwidth and values to be communicated. Bösen adopts Stale Synchronous Parallel for parameter synchronization, and in addition, selectively propagates parameter updates and fresh parameter values in a rate-limited fashion to reduce staleness in parameter values without congesting the network. The rate-limited communication prioritizes parameter updates based on the relative magnitude of the change to allocate the limited network bandwidth to the most important messages. During machine learning training, algorithmic hyperparameters, such as step size, need to be tuned to adapt to the changing synchronization frequency, in order to take full advantage of the frequent updates. To our knowledge, Bösen is the first distributed implementation of Adaptive Revision [110], a principled step-size tuning algorithm tolerant of delays in distributed systems. Adaptive Revision achieves theoretical convergence guarantees by adaptively adjusting the step size to account for errors caused by delayed updates.

We demonstrate the effectiveness of managed communication on three applications: Matrix Factorization with SGD, Topic Modeling (LDA) with Gibbs sampling, and Multiclass Logistic Regression with SGD on an up to 1024 core compute cluster. Our experiments on Matrix Factorization show orders of magnitude of improvements in the number of iterations needed to achieve convergence, compared to the best hand-tuned fixed-schedule step size. Even with a delay-tolerant algorithm, Bösen’s managed communication still improves
the performance of SGD with Adaptive Revision.

3.1 The Bösen Parameter Server Architecture

Bösen is a parameter server with an ML-consistent, bounded-staleness parallel scheme and bandwidth-managed communication mechanisms. It realizes bounded staleness consistency, which offers theoretical guarantees for iterative convergent ML programs (unlike TAP) while enjoying high computation throughput that is better than BSP and close to TAP systems. Additionally, Bösen transmits model updates and up-to-date model parameters proactively without exceeding a bandwidth limit, while making better use of the bandwidth by scheduling the bandwidth budget based on the contribution of the messages to algorithm progress — thus improving per-data-sample convergence compared to an agnostic communication strategy.

Bösen PS consists of a client library and parameter server partitions (Figure 3.1); the former provides the Application Programming Interface (API) for reading/updating model parameters, and the latter stores the master copy of the shared model parameters. In terms of usage, Bösen closely follows other key-value stores: once an ML program process is linked against the client library, any thread in that process may read/update model parameters concurrently. The user runs a Bösen ML program by invoking as many server partitions, and ML application compute processes (which use the client library) as needed, across multiple machines.

Data Abstraction and Bounded Staleness Consistency

Data Abstraction. Bösen represents model parameters and other values that need to be shared among the distributed compute processes as key-value pairs. The application program create different tables to store values of different types, e.g., floating-point vs. fixed-point, different precision, dense vs. sparse, etc.. In order to exploit locality in ML applications and thus amortize the overhead of concurrent operations, the parameters in a table are organized into rows — a row is a set of parameters that are usually accessed together. An
application program may create as many tables as needed, and each table can use a different data structure for its rows. Bösen provides commonly used row types, such as dense vectors and sparse maps, for convenience while supporting application-defined row types.

Table 3.1 shows the main APIs for reading and modifying shared values. To read or modify any shared value, the application program first obtains a handle to the corresponding table. Using Bösen’s API, the application program may read or modify a single value or a row of values. Additionally, Bösen supports user-defined “stored procedures” to be executed on each server, which can be used to alter the default increment behavior of parameter updates (see Sec 3.2.3).

**Bounded Staleness Consistency.** Compared to general-purpose key-value stores, a distinctive feature of Bösen is its bounded staleness consistency model, which was proposed by Ho et al. and was referred to as stale synchronous parallel (SSP) [75]. The consistency model defines what value may be seen, i.e., what writes the value may contain, when a key is queried by the application program.

In Bösen, we refer to an application compute thread as a worker and a unit of computation performed by a worker as a clock tick. The amount of work contained in a clock tick is defined by the application. For example, it could be a single mini-batch or multiple mini-batches. A worker signals the end of a clock tick by calling Clock(). Bösen's bounded staleness consistency model accepts an application-defined staleness threshold $S$ and en-
sures that a worker at the $t$-th clock tick may observe a parameter $a$'s value if and only if $a$ contains all updates from the $(t - S - 1)$-th and earlier clock ticks across all workers. We say that a parameter value is too stale with respect to the worker’s logical time if the value is missing updates from those clock ticks. The worker thread is blocked upon calling `Get()` or `GetRow()` until a sufficiently fresh parameter value is available.

**Parameter Synchronization.** Bösen clients cache parameter values and buffer parameter updates locally and transmit the buffered updates and fetch fresh parameter values after all application compute threads complete a clock tick. All worker threads in a compute process share a parameter cache, and buffering allows updates generated by different workers over many clock ticks to be coalesced to reduce communication volume. In order to reduce communication volume, earlier SSP implementations, such as LazyTable [44, 75], transmit only parameter updates that are $S$ clocks old and fetch a fresh parameter value only when the locally cached value is too stale. In contrast, Bösen transmits all buffered updates and fetches a new value for all parameters needed by this compute process at the end of a clock tick. An exemplary execution of 5 workers under Bounded Staleness is depicted in Fig 3.2.

Machine learning applications typically perform multiple mini-batches within in one clock tick (i.e., local buffering), especially when each mini-batch is computationally inexpensive, so parameter synchronization does not become a significant overhead. When the staleness threshold is set to 0, bounded staleness consistency reduces to the classic BSP model. The BSP model guarantees all updates computed in previous clock ticks are visible.
A positive staleness threshold allows the next clock tick to begin without having to wait for communication to finish, overlapping communication with computation. The bounded staleness model enjoys BSP-like ML execution guarantees, theoretically explored by Ho et al. and Wei et al. [49, 75]. Our experiments used a staleness threshold of 2 unless otherwise mentioned, which has been reported to be effective by Cui et al. [44].

3.1.1 System Architecture

This section describes Bösen’s system architecture and focuses on its realization of the bounded staleness consistency. The system described in this section sufficiently ensures the consistency guarantees without communication management. Bounded staleness consistency without communication management serves as our baseline in evaluation and is referred to as “Bounded Staleness” in Section 3.3.

Client Library

The client library provides access to the model parameters on the server partitions, using cache for faster access while cooperating with server processes in order to maintain consistency guarantees and manage bandwidth. This is done through three components: (1) a parameter cache that caches a partial or complete image of the model at the client, in order to serve read requests made by application compute threads; (2) an update buffer that buffers updates applied by compute threads via Inc() and RowInc(); (3) a group of client communication threads (distinct from compute threads) that perform synchronization of the local model cache and buffered updates with the servers’ master copies, while the compute threads executes the application algorithm.

The parameters cached at a client are hash partitioned among the client communication threads. Each client communication thread needs to access only its own parameter partition when reading the computed updates and applying up-to-date parameter values to minimize lock contention. The client parameter cache and update buffer allow concurrent reads and writes from worker threads, and similar to [45], the cache and buffer use static data structures and pre-allocate memory for repeatedly accessed parameters to minimize the overhead of maintaining a concurrent hash table.

In each compute process, locks are needed for shared access to parameters and buffered update entries. In order to amortize the runtime cost of concurrency control, we allow applications to define parameter key ranges called rows (as noted above). Parameters in the same row share one lock for accesses to their parameter caches, and one lock for accesses to their update buffers.

When serving read requests (Get() and RowGet()) from worker threads, the client parameter cache is searched first, and a read request is sent to the server processes only if either the requested parameter is not in the cache or the cached parameter’s staleness is not within the staleness threshold. The reading compute thread blocks until the parameter’s staleness is within the threshold. When writes are invoked, updates are inserted into the
update buffer, and, optionally, the client’s own parameter cache is also updated.

Once all compute threads in a client process have called \texttt{Clock()} to signal the end of a unit of work (e.g., a clock tick), the client communication threads release buffered model updates to servers. Note that buffered updates may be released sooner under managed communication if the system detects spare network bandwidth to use.

**Server Partitions**

The master copy of the model’s parameters is hash partitioned, and each partition is assigned to one server thread. The server threads may be distributed across multiple server processes and physical machines. As model updates are received from client processes, the addressed server thread updates the master copy of its model partition. When a client read request is received, the corresponding server thread registers a callback for that request; once a server thread has applied all updates from all clients for a given unit of work, it walks through its callbacks and sends up-to-date model parameter values.

**Ensuring Bounded Staleness**

Bounded staleness is ensured by coordination of clients and server partitions using \textit{clock messages}. On an individual client, as soon as all updates generated before and in the \( t \)-th clock tick are sent to server partitions and no more updates before or in that clock tick can be generated (because all compute threads have advanced beyond that clock), the client’s communication threads send a client clock message to each server partition, indicating “all updates generated before and in clock \( t \) by this client have been made visible to this server partition” (assuming reliable, ordered message delivery).

After a server partition sends out all dirty parameters modified in clock \( t \), it sends a server clock message to each client communication thread, indicating ‘all updates generated before and in clock \( t \) in the parameter partition have been made visible to this client’. Upon receiving such a clock message, the client communication thread updates the age of the corresponding parameters and permits the relevant blocked compute threads to proceed on reads, if any.

**Fault Tolerance**

Bösen provides fault tolerance by checkpointing the server model partitions; in the event of failure, the entire system is restarted from the last checkpoint. A valid checkpoint contains the model state \textit{strictly} right after clock \( t \) — the model state includes all model updates generated before and during clock \( t \), and excludes all updates after the \( t \)-th \texttt{Clock()} call by any worker thread. With bounded staleness, clients may asynchronously enter new clocks and begin sending updates; thus, whenever a checkpointing clock event is reached, each server model partition will copy-on-write protect the checkpoint’s parameter values until that checkpoint has been successfully copied externally. Since taking a checkpoint can be slow, a checkpoint will not be made every clock or even every few clocks. A good es-
timate of the amount of time between taking checkpoints is \( \sqrt{\frac{2T_sT_f}{N}} \) \[168\], where \( T_s \) is the meantime to save a checkpoint, there are \( N \) machines involved and \( T_f \) is the meantime to failure (MTTF) of a machine, typically estimated as the inverse of the average fraction of machines that fail each year.

As Bösen targets offline batch training, restarting the system (disrupting its availability) is not critical. With tens or hundreds of machines, such training tasks typically complete in hours or tens of hours. Considering the MTTF of modern hardware, it is not necessary to create many checkpoints, and the probability of restarting is low. In contrast, a replication-based fault tolerance mechanism inevitably costs \( 2x \) or even more memory on storing the replicas and additional network bandwidth for synchronizing them.

### 3.2 Managed Communication

Bösen’s client library and server partitions feature a communication manager whose purpose is to improve ML progress per epoch (i.e., data pass) through careful use of network bandwidth in communicating model updates/parameters. Communication management is complementary to consistency management; the latter prevents worst-case behavior from breaking ML consistency (correctness), while the former improves convergence time (speed).

The communication manager has two objectives: (1) communicate as many updates per second as possible (full utilization of the bandwidth budget) without overusing the network (which could delay update delivery and increase message processing computation overhead); and (2) prioritize more important model updates to improve ML progress per epoch. The first objective is achieved via bandwidth-driven communication with rate limiting, while the second is achieved by choosing a proper prioritization strategy.

#### 3.2.1 Bandwidth-Driven Communication

Similar to the leaky bucket model, the Bösen communication manager models the network as a bucket that leaks bytes at a certain rate and the leaky rate corresponds to the node’s bandwidth consumption. Thus the leaky rate is set to the given bandwidth budget to constrain the average bandwidth consumption. In order to fully utilize the given bandwidth budget, the communication manager permits communication of updates or updated parameters whenever the bucket becomes empty (and thus communication may happen before the completion of a clock tick). The communication manager keeps track of the number of bytes sent last time to monitor the state of the bucket. In our prototype implementation, the communication threads periodically query the communication manager for opportunities to communicate. The size of each send is limited by the size of the bucket (referred to as “queue size”) to control its burstiness.

**Coping with network fluctuations.** In real cloud data centers with multiple users, the available network bandwidth may fluctuate and fail to live up to the bandwidth budget \( B \). Hence, the Bösen communication manager regularly checks to see if the network is overused by monitoring how many messages were sent without acknowledgment in a recent time win-
dow (i.e., message non-delivery). If too many messages fail to be acknowledged, the communication manager assumes that the network is overused, and waits until the window becomes clear before permitting new messages to be sent.

**Update quantization.** As we discussed in Sec. 2.4.3, there has been a growing body of work in recent years that uses reduced precision and quantization to represent floating-point numbers to reduce the overhead of communicating model updates. Bösen applications have the option to use IEEE half-precision 16-bit floating-point numbers for communication, reducing bandwidth consumption in half compared to 32-bit floats. The lost information often has a negligible impact on progress per epoch.

### 3.2.2 Update Prioritization

Bösen spends available bandwidth on communicating information that contributes the most to convergence. For example, gradient-based algorithms (including Logistic Regression) are iterative-convergent procedures in which the fastest-changing parameters are often the largest contributors to solution quality — in this case, we prioritize communication of fast-changing parameters, with the largest-magnitude changes going out first. When there is an opportunity for communication due to spare bandwidth, the server or client communication threads pick a subset of parameter values or updates to send. The prioritization strategy determines which subset is picked at each communication event. By picking the right subset to send, the prioritization strategy alters the communication frequency of different parameters, effectively allocating more network bandwidth to more important updates. It should be noted that the end-of-clock communication needs to send all up-to-date parameters or updates older than a certain clock number to ensure the consistency guarantees.

Bösen’s bandwidth manager supports multiple prioritization strategies. The simplest possible strategies are **Randomized**, where communications threads send out randomly-chosen rows and **Round-Robin**, where communication threads repeatedly walk through the rows following a fixed order, and sends out all non-zero updates or updated parameters encountered. These strategies are baselines; better strategies prioritize according to significance to convergence progress. We study the following two better strategies.

**Absolute Magnitude prioritization.** Updates/parameters are sorted by their accumulated change in the buffer, $|\delta|$.

**Relative Magnitude prioritization.** Same as absolute magnitude, but the sorting criteria is $|\delta/a|$, i.e., the accumulated change normalized by the current parameter value, $a$. For some ML problems, relative change $|\delta/a|$ may be a better indicator of progress than absolute change $|\delta|$. In cases where $a = 0$ or is not in the client parameter cache, we fall back to absolute magnitude prioritization.
3.2.3 Adaptive Step Size Tuning

Many data-parallel ML applications use the stochastic gradient descent (SGD) algorithm, whose updates are gradients multiplied by a scaling factor, referred to as “step size” and typically denoted as $\eta$. The update equation is thus:

$$A^{(t)} = A^{(t-1)} + \sum_{p=1}^{P} \eta_p^{(t-1)} \nabla (A^{(t-1)}, D_p).$$ (3.1)

The SGD algorithm’s performance is very sensitive to the step size used. Early distributed SGD applications (i.e., GraphLab’s SGD MF, MLlib’s SGD LR, etc.) apply the same step size for all dimensions and decay the step size each epoch according to a fixed schedule. Achieving ideal algorithm performance requires a great amount of manual tuning to find a good initial step size. Adaptive gradient algorithms adaptively adjust the step size, reducing sensitivity to the initial step size $\eta^{(1)}$ and achieving good algorithm performance using any initial step size from a reasonable range. In order to achieve fast convergence, AdaGrad [54] adjusts the step size for each dimension differently to perform small updates to parameters associated with frequently occurring features and large updates to parameters associated with rare features. Adaptive Revision (or AdaRevision) [110] extends AdaGrad for distributed training. In addition to scaling gradients based on update frequency, AdaRevision scales down gradients that are computed using stale parameter values, to mitigate network delays in asynchronous data-parallel training. While other adaptive step size algorithms, such as RMSprop [12] and Adam [92], are probably more widely used for training DNNs, we focus on AdaRevision here because scaling up updates that are received sooner amplifies the effect of quicker communication. Note that other adaptive step size algorithms can also be implemented by applications when needed using Bøsen primitives described below.

Compared to regular SGD, an AdaRevision implementation additionally maintains an accumulated sum of historical gradients for each parameter; a gradient atomically updates the parameter and the accumulated sum. When a parameter is read out of the parameter store for computation, a snapshot of the accumulated sum is taken and returned along with the parameter value. A client will compute a gradient using that parameter, and then apply it back to the parameter store — when this happens, the snapshot associated with that parameter is also supplied. The difference between the snapshot value and the latest parameter value indicates the timeliness of the update, and is used to adjust the step size: the longer the updates are delayed, the smaller the step size, so that long-delayed gradients do not jeopardize model quality. Our implementation stores the accumulated sum of historical gradients on the server partitions, and thus the updates are only applied on the server, while the client parameter cache is made read-only to the compute threads.

Whereas a naive implementation of AdaRevision might let clients fetch the accumulated sum (which is generally not needed for computing gradients) along with the parameters from the server (and send the sum back along with the computed gradients), Bøsen instead supports parameter versioning to reduce the communication overhead. The server maintains a version number for each parameter row, which is incremented every time the row is updated. The version number is sent to clients along with the corresponding parameters.
and stored in the client’s parameter cache. The update computed (by a worker) for parameter $i$ is tagged with the version number of parameter $i$ in the parameter cache. The updates tagged with the same version number are aggregated via addition as usual, but updates with different version numbers are stored separately. The use of version number to indicate the timeliness of writes is similar to optimistic concurrency control.

The AdaRevision algorithm is implemented as a user-defined stored procedure (UDF) on the server. The user-defined stored procedure contains a set of user-implemented functions that are invoked at various events to control the server’s behavior. Most notably, the UDF takes snapshots of the accumulated sum of the historical gradients and increments the version number upon sending parameters to clients and computes the step size when applying gradients. In order to bound the overall memory usage, the UDF imposes an upper limit on the number of snapshots that can be kept. Bandwidth-triggered communication is canceled upon exceeding this limit. The snapshots are freed when no client cache still contains this version of the parameter.

We demonstrate the importance of step size tuning and effectiveness of adaptive revision using the SGD MF application on the Netflix dataset, with rank = 50, using one node in the PRObE Susitna cluster (see Sec 3.3). As shown in Fig. 3.3, we compared adaptive revision (AdaRev) with Multiplicative Decay (MultiDecay) using various initial step sizes. We also ran GraphLab’s SGD MF using its synchronous engine (the asynchronous engine converges slower per epoch) with a range of initial step sizes from $1e^{-4}$ to $1e^{-6}$ and showed its best convergence result.

Firstly, we observed that multiplicative decay is sensitive to the initial step size. Changing the initial step size from $6e^{-5}$ to $1.2e^{-5}$ reduces the number of epochs needed to reach

---

**Figure 3.3:** Compare Bösen’s SGD MF w/ and w/o adaptive revision with GraphLab SGD MF. Eta denotes the initial step size. Multiplicative decay (MultiDecay) used its optimal initial step size.
Table 3.2: Datasets used in evaluation. Data size refers to the input data size. Workload refers to the total number of data samples in the input data set.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Workload</th>
<th>Description</th>
<th>Data Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Netflix</td>
<td>100M ratings</td>
<td>480K users, 18K movies, rank=400</td>
<td>1.3GB</td>
</tr>
<tr>
<td>NYTimes</td>
<td>99.5M tokens</td>
<td>300K documents, 100K words 1K topics</td>
<td>0.5GB</td>
</tr>
<tr>
<td>ClueWeb10%</td>
<td>10B tokens</td>
<td>50M webpages, 160K words, 1K topics</td>
<td>80GB</td>
</tr>
<tr>
<td>ImageNet5%</td>
<td>65K samples</td>
<td>1000 classes, 21K of feature dimensions</td>
<td>5.1GB</td>
</tr>
</tbody>
</table>

Table 3.3: Descriptions of ML models and evaluation datasets. The overall model size is thus # Rows multiplied by row size.

<table>
<thead>
<tr>
<th>Application &amp; Dataset</th>
<th># Machines</th>
<th>Bandwidth Budgets</th>
<th>Queue Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGD MF, Netflix</td>
<td>8 (N)</td>
<td>200Mbps, 800Mbps</td>
<td>100, 100</td>
</tr>
<tr>
<td>LDA, NYTimes</td>
<td>16 (N)</td>
<td>320Mbps, 640Mbps, 1280Mbps</td>
<td>5000, 500</td>
</tr>
<tr>
<td>LDA, ClueWeb10%</td>
<td>64 (N)</td>
<td>800Mbps</td>
<td>5000, 500</td>
</tr>
<tr>
<td>MLR, ImageNet5%</td>
<td>4 (S)</td>
<td>100Mbps, 200Mbps, 1600Mbps</td>
<td>1000, 500</td>
</tr>
</tbody>
</table>

Table 3.4: Bösen system and application configurations. N - cluster Nome, S - cluster Susitna. The queue size (in number of rows) upper bounds the send size to control burstiness; the first number denotes that for client and the second for server. LDA experiments used hyper-parameters $\alpha = \beta = 0.1$. SGD MF and MLR uses an initial learning rate of 0.08 and 1 respectively.

a training loss of $1e8$ by more than $3\times$. However, convergence with adaptive revision is much more robust, and the difference between the initial step size of 0.08 and 0.4 is negligible. Secondly, we observed that SGD MF under adaptive revision converges $2\times$ faster than using multiplicative decay with the optimal initial step size that our manual parameter tuning could find. Even though GraphLab also applies multiplicative decay to its step size, it does not converge well.

The adaptive revision algorithm becomes more effective when scaling the SGD application as it adapts the step size to tolerate the communication delay. An experiment (not shown) using 8 Susitna nodes shows that adaptive revision reduces the number of epochs to convergence by $10\times$.

3.3 Evaluation

We evaluated Bösen using three real machine learning applications, matrix factorization (MF) solved by SGD, latent Dirichlet allocation (LDA) solved by collapsed Gibbs sampling,
and multiclass logistic regression (MLR) also solved by SGD.

**Cluster setup**: Most of our experiments were conducted on PRObE Nome [62], consisting of 200 computers running Ubuntu 14.04. Our experiments used different numbers of computers, varying from 8 to 64. Each machine contains $4 \times$ quad-core AMD Opteron 8354 CPUs (16 physical cores per machine) and 32GB of RAM. The machines were distributed over multiple racks and connected via a 1 Gb Ethernet and 20 Gb Infiniband. A few experiments were conducted on PRObE Susitna [62]. Each machine contains $4 \times 16$-core AMD Opteron 6272 CPUs (64 physical cores per machine) and 128GB of RAM. The machines are distributed over two racks and connected to two networks: 1 GbE and 40 GbE. In both clusters, every machine is used to host Bösen server, client library, and worker threads (i.e., servers and clients are collocated and evenly distributed).

**ML algorithm setup**: In all ML applications, we partition the data samples evenly across the workers. Unless otherwise noted, we adopted the typical BSP configuration and configured 1 logical clock tick to be 1 pass through the worker’s local data partition$^1$. The ML models and datasets are described in Table 3.3, and the system and application configurations are described in Table 3.4.

**Performance metrics**: Our evaluation measures performance as the absolute convergence rate on the training objective value; that is, our goal is to reach convergence to an estimate of the model parameters that best represent the training data (as measured by the training objective value) in the shortest time.

Bösen is executed under different modes in this section:

**Single Node**: The ML application is run on one shared-memory machine linked against one Bösen client library instance with only consistency management. The parameter cache is updated upon write operations. Thus updates become immediately visible to compute threads. It represents a gold standard when applicable. It is denoted as “SN”.

**Linear Scalability**: It represents an ideal scenario where the single-node application is scaled out, and linear scalability is achieved. It is denoted as “LS”.

**Bounded Staleness**: Bösen is executed with only consistency management enabled, and communication management is disabled. It is denoted as “BS”.

**Bounded Staleness + Managed Communication**: Bösen is executed with both consis-

$^1$In one clock we compute parameter updates using each of the $N$ data samples in the dataset exactly once, regardless of the number of parallel workers. With more workers, each worker will touch fewer data samples per data pass.
tency and communication management enabled. It is denoted as “MC-X-P”, where X denotes the per-node bandwidth budget (in Mbps), and P denotes the prioritization strategy: "R" for Randomized, "RR" for Round-Robin, and "RM" for Relative-Magnitude.

**Bounded Staleness + Fine-Grained Clock Tick Size:** Bösen is executed with only consistency management enabled, communication management is disabled. In order to communicate updates and model parameters more frequently, a full pass over data is divided into multiple clock tick. It is denoted as “BS-X”, where X is the number of clock ticks that constitutes a data pass.

Unless otherwise mentioned, we used a staleness threshold of 2 and we found that although bounded staleness converges faster than BSP, changing the staleness threshold does not affect average-case performance as the actual staleness is usually 1 due to the eager end-of-clock communication (Section 3.1). The network waiting time is small enough that a staleness threshold of 2 achieves no blocking. The bounded staleness consistency model allows computation to proceed during synchronization. As long as the workload is balanced and synchronization completes within one clock tick of computation (which is typically the case), the network waiting time can be completely hidden.

### 3.3.1 Communication Management

In this section, we show that the algorithm performance improves with more immediate communication of updates and model parameters. Moreover, proper bandwidth allocation based on the importance of the messages may achieve better algorithm performance with less bandwidth consumption. To this end, we compared managed communication with non-managed communication (i.e., only the consistency manager is enabled). The communication management mechanism was tested with different per-node bandwidth budgets (see Table 3.4) and different prioritization strategies (Section 3.2.2). Each node runs the same number of client and server communication threads and the bandwidth budget is evenly divided among them.

**Effect of increasing bandwidth budget.** Under Bösen’s communication management, increasing bandwidth budget permits more immediate communication of model updates and parameters and thus improves algorithm performance (higher convergence per epoch, i.e., data pass) given a fixed prioritization policy. We demonstrate this effect via the MF and LDA experiments (Fig. 3.4). First of all, we observed that enabling communication management significantly reduces the number of epochs needed to reach convergence (objective value of 2e7 for MF and −1.022e9 for LDA). In MF, communication management with bandwidth budget of 200Mbps reduces the number of epochs needed to reach 2e7 from 64 (BS) to 24 (MC-200-R). In LDA, a bandwidth budget of 320Mbps reduces the number of epochs to convergence from 740 (BS) to 195 (MC-320-R). Secondly, increasing the bandwidth budget further reduces the number of epochs needed. For example, in LDA, increasing the bandwidth budget from 320Mbps (MC-320-R) to 640Mbps (MC-640-R) reduces the number of epochs needed from 195 to 120.
**Effect of prioritization.** As shown Fig. 3.4b, in the case of LDA, prioritization by Relative-Magnitude (RM) consistently improves upon Randomization (R) when using the same amount of bandwidth. For example, with 320Mbps of per-node bandwidth budget MC-320-RM reduces the number of epochs needed to reach $-1.022e9$ from 195 (MC-320-R) to 145.

Relative-Magnitude prioritization improves upon Randomized prioritization as it differentiates updates and model parameters based on their significance to algorithm performance. It allocates network bandwidth accordingly and communicates different updates and model parameters at different frequencies. Fig. 3.5 shows the CDFs of communication frequency of LDA’s model parameters, under different policies. For the NYTimes dataset, we observed that Relative-Magnitude and Absolute-Magnitude prioritization achieve similar effect, where a small subset of keys are communicated much more frequently. Random and Round-Robin achieve similar effect where all keys are communicated at roughly the
Prioritization appears to be less effective for MF. The server UDF computes the step size which scales the gradient, altering the gradient by up to orders of magnitude. Since the adaptive revision algorithm tends to [110] apply a larger scaling factor for smaller gradients, the raw gradient magnitude is a less effective indicator of significance.

**Overhead of communication management and absolute convergence rate.** Under managed communication, the increased volume of messages incurs noticeable CPU overheads due to sending and receiving the messages and serializing and deserializing the content. Computing importance also costs CPU cycles. Fig. 3.6 presents the per-epoch runtime and network bandwidth consumption corresponding to Fig. 3.4. For example, enabling communication management with a 200 Mbps bandwidth budget (MC-200-R) incurs a 12% per-epoch runtime overhead.

However, the improved algorithm performance significantly outweighs such overheads and results in much higher absolute convergence rate in wall clock time, as shown in Fig. 3.7 (MF and MLR) and Fig. 3.10a. For example, for MF, we observed a $2.5 \times$ speedup in absolute convergence rate using bandwidth budget of 800 Mbps and Relative-Magnitude prioritization compared the bounded staleness baseline.

**Comparison with Yahoo!LDA.** We compare Bösen LDA with the popular Yahoo!LDA using the NYTimes and 10% of the ClueWeb data set, using 1GbE and 20 Gb Infiniband, respectively. The former is plotted in Fig. 3.8. Yahoo!LDA employs a parameter server architecture that’s similar to Bösen’s, but uses total asynchronous parallelization. The compute threads of Yahoo!LDA process roughly the same number of data points as Bösen’s. Each Yahoo!LDA worker (node) runs one synchronizing thread that iterates over and syn-

---

2 On a skewed dataset, it’s possible to observe a skewed communication frequency distribution even with Randomized or Round-Robin policy when some words appear much more frequently than others. Even then the prioritization scheme can still alter the communication frequency to prioritize the most important parameters.
Figure 3.6: Overhead of communication management: time per data pass and average bandwidth consumption. Note that while managed communication consumes high network bandwidth and takes longer to perform a mini-batch, it significantly reduces the number of epoches needed to reach the target objective function value (see Fig. 3.4) and thus improves the wall clock time to convergence (see Fig. 3.7).

chronizes all cached parameter in a predefined order. We observed that Bösen significantly outperformed Yahoo!LDA on the NYTimes dataset, but converged at similar rate on the ClueWeb10% data set.

In summary, by making full use of the 800Mbps and 640Mbps bandwidth budget, com-
Figure 3.7: Absolute convergence rate under managed communication

Communication management with Randomized prioritization improved the time to convergence of the MF and LDA application by $2.5 \times$ and $2.8 \times$ in wall clock time and $5.3 \times$ and $6.1 \times$ in number of epochs, compared to a bounded staleness execution. Relative-Magnitude prioritization further improves the convergence time of LDA by 25%. Communication management with bandwidth budget of 200Mbps and Relative-Magnitude prioritization improved the convergence time of MLR by $2.5 \times$.

### 3.3.2 Comparison with Clock Tick Size Tuning

Another way of reducing parallel error on a BSP or bounded staleness system is to divide a full data pass into multiple clock ticks to achieve more frequent synchronization, while properly adjusting the staleness threshold to ensure the same staleness bound. This approach is similar to mini-batch size tuning in ML literature. In this section, we compare
**Figure 3.8:** Compare Bösen LDA with Yahoo!LDA on NYTimes Data

**Figure 3.9:** Comparing Bösen with simply tuning clock tick size: convergence per epoch

Bösen’s communication management with application-level clock tick size tuning via the LDA application and the result is plotted in Fig 3.10. For each number of clock ticks per data pass, we adjust the staleness threshold so all runs share the same staleness bound of 2 data passes.

Firstly, from Fig. 3.10a we observe that as the clock tick size halves, the average bandwidth usage over the first 280 epochs doubles but the average time per epoch doesn’t change significantly. From Fig. 3.9, we observe that the increased communication improves the algorithm performance. Although simply tuning clock tick size also improves algorithm behavior, it doesn’t enjoy the benefit of prioritization. For example, MC-640-RM used only
Figure 3.10: Comparing Bosen with simply tuning clock tick size

63% of the bandwidth compared to BS-8 but converged 28% faster. The difference is due to careful optimization which cannot be achieved via application-level tuning.

3.4 Summary

While tolerance to bounded staleness reduces communication and synchronization overheads for distributed machine learning algorithms and thus improves system throughput, the accumulated error may, sometimes heavily, harm algorithm performance and result in slower convergence rate. More frequent communication reduces staleness and parallel error and thus improves algorithm performance but it is ultimately bound by the physical network capacity. This paper presents a communication management technique to maximize the communication efficiency of bounded amount of network bandwidth to improve
algorithm performance. Experiments with several ML applications on over 1000 cores show that our technique significantly improves upon static communication schedules and demonstrate an up-to-5× speedup relative to a well implemented bounded staleness system.

Our prototype implementation has certain limitations. While a production system should address these limitations, our evaluation nevertheless demonstrates the importance of managing network bandwidth. Our implementation assumes all nodes have the same inbound and outbound bandwidth and each node’s inbound/outbound bandwidth is evenly shared among all nodes that it communicates with. Such assumption is broken in a hierarchical network topology typically seen in today’s data centers, leading to under-utilized network bandwidth. Although update magnitude serves as a good indicator of update importance for some applications, there are cases, such as when stored procedures are used, where it may be insufficient. Future research should look into exploiting more application-level knowledge and actively incorporating server feedbacks to clients.
Chapter 4

Application-Specific Computation Scheduling Case Study

A key problem of data parallelism is that it is not equivalent to serial execution because a worker may not observe the parameter updates that are produced in parallel by other workers. Compared to serial execution, under data parallelism, a worker computes parameter updates using a stale version of model parameter values, violating data dependence.

Non-serializable execution often leads to slower algorithm convergence and lower model quality. Therefore data parallelism is not always the best parallelization method. We can understand the effect of such non-serializable parallelization from two perspectives. First, for stochastic gradient descent (SGD), synchronous data parallelism over \( K \) workers is equivalent to sequential SGD using a mini-batch size of \( K \) times larger. Mini-batch size is an SGD hyperparameter, and a mini-batch size that is too large often requires more data passes to reach the same model quality and may also lead to lower model performance on unseen data. Previous work reported this effect for both traditional ML models [90] and neural networks [77, 89]. Second, generally speaking, non-serializable parallelization is an erroneous execution of the sequential algorithm, where parameter values contain error due to conflicting accesses. Intuitively, the error's magnitude increases when more workers are used and decreases when workers synchronize more frequently. Thanks to ML algorithms' tolerance to bounded error [75, 127], the erroneous execution may still produce an acceptable model, but the algorithm's convergence rate and model quality degrades as the error increases [75, 90, 157]. Large mini-batch size or synchronization once per multiple mini-batches is common in distributed training in order to amortize synchronization overhead. This is especially common for traditional ML models where per-data-sample computation is light.

Some machine learning training programs exhibit a sparse parameter access pattern where each training data sample reads and updates only a subset of the model parameters. This sparse parameter access pattern may allow parallel mini-batch computation to be scheduled in a way that eliminates conflicting accesses and thus retains serializability. In this chapter, we present two machine learning applications, LDA for topic modeling,
and SGD matrix factorization, that leverage this opportunity to improve distributed training efficiency. The LDA training application is implemented based on Bösen to leverage its efficient distributed shared memory abstraction and SGD matrix factorization is implemented on Apache Spark to evaluate the RDD abstraction for distributed training. While implementations based on an efficient Parameter Server system (Bösen) achieve high performance, they also require substantial programmer effort. On the other hand, Spark’s high-level programming interface, especially the Python API, substantially reduces programmer effort, but the Spark implementation suffers significant performance overhead because the immutable RDD abstraction and the map-reduce execution model is inefficient for the rapidly updated model parameters in machine learning training.

4.1 LightLDA: Scheduling Computation for Latent Dirichlet Allocation

4.1.1 Introduction

Topic models have been widely applied in text mining, network analysis and genetics, and other domains [19, 26, 137, 167, 177]. With the rapid growth of data size, it has become crucial to scale topic models, particularly the Latent Dirichlet Allocation (LDA) model [26], to web-scale corpora. Web-scale corpora are significantly more complex than smaller, well-curated document collections, and thus require a high-dimensional parameter space featuring up to millions of topics and vocabulary words and hence trillions of parameters, in order to capture long-tail semantic information that would otherwise be lost when learning only a few thousands of topics [156].

LightLDA [171] proposes a new sampling algorithm that reduces the per-token sampling complexity of commonly used collapsed Gibbs sampling algorithm from $O(K)$ to $O(1)$, where $K$ is the number of topics. The new LightLDA sampler exhibits a similar parameter access pattern to the classic collapsed Gibbs sampling algorithm. By analyzing the per-token parameter access pattern, we design a static scheduling algorithm that schedules independent computation with respect to model parameter access to improve the convergence rate upon data parallelism and enable training large models that do not fit into a single machine. Thanks to its fast sampling algorithm and the sophisticated distributed implementation, LightLDA was able to train a LDA model with $K = 1$ million topics in less than 2 days on a training corpus with a vocabulary size of 1 million (1 trillion shared parameters and 200 billion tokens) using only 24 CPU machines (480 cores in total). To the best of my knowledge, this was the largest topic model reported in publications in 2015. The previous state-of-the-art topic model instance has 2000 topics and was trained on a training corpus with a vocabulary size of 5 million (10 billion shared parameters), which used 6000 machines (60000 cores in total) for about one day.
4.1.2 Background: Latent Dirichlet Allocation and Gibbs Sampling

In this section, we briefly review the Latent Dirichlet Allocation (LDA) [26] model. Specifically, LDA assumes the following generative process for each document in a corpus:

- \( \varphi_k \sim \text{Dirichlet}(\beta) \): Draw word distribution \( \varphi_k \) per topic \( k \).
- \( \theta_d \sim \text{Dirichlet}(\alpha) \): Draw topic distribution \( \theta_d \) per document \( d \).
- \( n_d \sim \text{Poisson}(\gamma) \): For each document \( d \), draw its length \( n_d \) (i.e., the number of tokens it contains).
- For each token \( i \in \{1, 2, \ldots, n_d\} \) in document \( d \):
  - \( z_{di} \sim \text{Multinomial}(\theta_{di}) \): Draw the token’s topic.
  - \( w_{di} \sim \text{Multinomial}(\varphi_{z_{di}}) \): Draw the token’s word.

To find the most plausible topics in a corpus and document-topic assignments, one must infer the posterior distribution of latent variables in an LDA model, by using either a variational- or sampling-based inference algorithm. Both the widely used collapsed Gibbs sampler [68] and the new fast LightLDA sampler sequentially sample a topic for each token according to the statistics of the current topic assignment to tokens. The important statistics include \( n_{kd} \), which is the number of tokens in document \( d \) that are assigned to topic \( k \), \( n_{kw} \), which is the number of tokens with word \( w \) (across all documents) that are assigned to topic \( k \), and \( n_k \), which is the number of tokens (across all docs) assigned to topic \( k \). The counts \( n_{kd} \), \( n_{kw} \), and \( n_k \), are cached and updated during training, which we refer to model parameters. We refer to all \( n_{kd} \) as the document-topic table, all \( n_{kw} \) as the word-topic table, and all \( n_k \) as the topic summary. These statistics are updated after each new topic is sampled, and the new statistics are used for sampling the next topic.

4.1.3 Scheduling Computation

We make two observations regarding the LDA model and the Gibbs sampling algorithm:

- Each document is represented as a bag of words, regardless of the ordering of the tokens. While the tokens should be processed sequentially, the order in which the tokens are processed can be arbitrary.
- Processing each token reads and updates only a small subset of the counts in the word-topic table \( n_{w} \) and the document-topic table \( n_{d} \) according to the word and the document the token corresponds to.

Based these two observations, we may carefully partition the corpus and schedule computation to workers to ensure a schedule in which during any concurrent set of mini-batch processing, there is no conflicting access on the word-topic table and the document-topic table. This parallelization achieves a higher per-data-sample convergence rate compared to data parallelism and also bounds the memory footprint of each worker for locally caching model parameters and buffer updates. While such a parallel computation schedule could
still incur conflicting access on the topic summary, its effect on algorithmic convergence is negligible. Fig. 4.1 shows an example of a $3 \times 3$ partitioning of the dataset and its computation schedule.

In order to scale to extremely large datasets that do not fit in the memory of the distributed worker machines, our LightLDA implementation leverages external storage (in our case, hard disks) to store the partitioned datasets. The corpus is statically partitioned among workers by the document; within each document, the tokens are partitioned by word. The worker loads only the relevant data partition into memory and uses double buffering to hide the I/O overhead.

LightLDA is implemented as an application on top of the Bösen Parameter Server to leverage its shared memory abstraction and the Bounded Staleness consistency model. The word-topic table and the topic summary are stored in Bösen, and thus shared by all workers. However, note that static scheduling ensures non-conflicting parameter access on the word-topic table and thus LightLDA does not take advantage of Bösen’s communication management mechanism. Static computation scheduling also ensures a worker processes only tokens from appropriate documents so that the document-topic table is partitioned among the worker machines. The training dataset, along with the topic indicator for each token, is statically partitioned by the application program among the worker machines’ external storage and loaded for computation in a streaming fashion, as was just described above.

### 4.1.4 Evaluation

In this section, we demonstrate that LightLDA efficiently trains the largest LDA model on a large dataset within 2 days. We use a cluster that contains 24 machines, in which each machine has 20 physical cores and 256 GB of memory. The machines are connected using 1 Gbps Ethernet. The LDA model is trained on a proprietary Bing Web Chunk dataset, which has a vocabulary size of 1 million and contains 1.2 billion documents and 200 billion tokens.

Fig. 4.2 presents the LDA model’s log-likelihood, i.e., training objective function value, during the course of training. We observe that the using 24 machines, LightLDA achieves a
near 3× speedup compared to using 8 machines, that is, 60 hours at 24 machines matches the log-likelihood of 180 hours at 8 machines.

![Figure 4.2: LightLDA log-likelihood over time.](image)

Fig. 4.3 shows the breakdown of the execution time. Note that in the first 10 iterations, communication is a significant overhead. However, the communication overhead reduces over time as the model becomes more and more sparse, i.e., each word is only associated with a small number of topics. The high ratio of computation vs. communication demonstrates that the system design achieves high efficiency. Linear scaling of time to objective function value is perhaps too high a bar to expect, but in this case, at least LightLDA scales very well with additional parallel resources.

![Figure 4.3: LightLDA breakdown of per-iteration time.](image)
4.2 Distributing SGD Matrix Factorization using Apache Spark

4.2.1 Introduction

Apache Spark \[13, 173\] is a distributed computing system that implements the map-reduce \[51\] programming model and is attracting wide attention for commercial Big Data processing. Spark achieves high throughput compared to the previously dominant open-source map-reduce implementation Apache Hadoop \[4\] by retaining data in main memory whenever possible and possibly through better implementation of its operations (such as reusing JVM across jobs, etc.). Moreover, it allows application developers to program in a declarative language because it employs a DAG scheduler that executes the inferred execution plan. It is said that the DAG scheduler greatly improves the programmer productivity as it relieves programmers from maintaining the complex dependencies among different tasks and scheduling them and also fault tolerance.

Spark has been perceived as a suitable choice for iterative algorithms, including ML training, as it avoids the costly disk I/O between iterations employed by previous map-reduce systems. Moreover, its declarative programming language (and DAG scheduler) may allow faster application development. However, as Spark does not apply any ML-specific optimizations mentioned above, it remains a question whether Spark may achieve throughput comparable to specialized distributed algorithm implementations or ML-oriented systems. In order to explore this question, we implemented a well-known parallel ML algorithm – Distributed Stochastic Gradient Descent for Matrix Factorization \[60\] on Spark and compared it with alternative implementations. We found that our PySpark implementation suffers significant runtime penalty (226\(\times\) slower than our specialized implementation in C++) and scales poorly to a larger number of machines or larger datasets, justifying research on machine-learning-optimized training frameworks.

4.2.2 Background: Spark and SGD Matrix Factorization

Spark

Spark organizes data into Resilient Distributed Datasets (RDD). An RDD is a read-only, partitioned collection of records. RDDs can only be created (“written”) through deterministic, coarse-grained operations on either 1) data in persistent storage or 2) other RDDs \[174\]. Such operations are referred to as transformations, which include map, filter, and join. Spark maintains the lineage of each RDD, i.e., how the RDD is created from data in persistent storage, so Spark doesn’t need to materialize the RDDs immediately, and any missing RDDs may be created through the deterministic operations according to its lineage. Another kinds of RDD operations are referred to as actions, which trigger computation that returns results to the Spark program. Examples include count and reduce.

Spark application developers write a driver program that defines RDDs and invokes operations on them. The Spark runtime consists of a master and many workers (i.e., executors).
The driver program is executed by the master, which commands the workers to execute the RDD operations accordingly. RDD transformations are lazily evaluated, i.e., they are not executed when they are encountered in code processing. Instead, the RDDs’ lineage graphs are recorded, and the associated transformations are executed only when they are needed for computing values returned to the application program (i.e., actions).

There are two types of RDD dependencies: narrow dependencies, where each parent RDD partition is needed by at most one child RDD partition, and wide dependencies, where multiple child partitions may depend on it. Narrow dependencies (e.g., map) allow pipelined execution on one cluster node. Wide dependencies (e.g., join) depend on many or all parent partitions and thus generally require shuffling. A spark scheduler groups as many pipelined transformations as possible into one stage. The boundary of each stage is shuffle operation. When an action is run, the scheduler builds a DAG of stages to execute from the RDD’s lineage graph.

Spark supports two mechanisms for communicating values to workers, which execute RDD operations. Firstly, the driver program may broadcast the value of a local variable to workers as a read-only copy. Such variables are referred to as broadcast variables. Additionally, the collect operation on an RDD creates a local variable in the driver program that consists of all identified records. Together collect and broadcast allow the driver to intervene and manage communication. Secondly, two RDDs may be joined by a field in their records as a key. Joined RDDs communicate information between workers without driver intervention by joining records from other RDDs that are in different machines, then partitioning the new RDD to cause information that used to be in different machines to land in the same machine. As data sizes scale, joins become the more efficient form of worker communication.

Spark also provides the Spark SQL library, which is built on top of RDDs and enables querying distributed datasets using the SQL language [23]. The main abstraction in Spark SQL is DataFrame, which is a distributed collection of data records with a schema. DataFrame can also be manipulated using RDD APIs besides SQL. Similar to RDD programs, Spark SQL applications are implemented by constructing a computation graph composed of pre-defined DataFrame operators. For many applications, the rich collection of pre-defined operators eliminate the need for user-defined functions (e.g., a map function on RDDs). This, along with the DataFrame schema, enables the query optimizer (i.e., Catalyst) to optimize the computation plan and generate efficient Java bytecode. However, machine learning programs often perform complex computation on each data record, which cannot be expressed using existing DataFrame operators and thus still rely on user-defined functions. Therefore, machine learning programs (at least the one that we study in this section) cannot take advantage of the Catalyst optimizer. Moreover, as DataFrame is built on top of RDDs, it uses RDD’s communication mechanisms. Due to these reasons, we did not observe meaningful performance improvements when using DataFrame compared to RDD, and thus we focus on RDD implementations in our study for its greater flexibility.
Matrix Factorization using SGD

Matrix factorization (MF) is a popular model used in recommender systems [93]. Given a large (and sparse) \( m \times n \) matrix \( V \) and a rank \( r \), the goal of MF is to find an \( m \times r \) matrix \( W \) and an \( r \times n \) matrix \( H \) such that \( V \approx WH \), where the quality of approximation is defined by an application-dependent loss function \( L \). In recommender systems, \( V \) represents a sparse user-item rating matrix, since any single user most likely rates only a subset of the items. We can predict a user’s interest on an item by predicting missing data within the sparse matrix using \( WH \). A commonly used loss function in recommender systems is the nonzero squared loss \( L_{NZSL} = \sum_{i,j: V_{ij} \neq 0} (V_{ij} - [WH]_{ij})^2 \) (missing entries are denoted as zeros).

MF is often solved as an optimization problem using Stochastic Gradient Descent (SGD) that minimizes the loss function (i.e., the objective function). Note that \( L_{NZSL} \) can be decomposed into the sum of local losses, i.e., \( L_{NZSL} = \sum_{i,j: V_{ij} \neq 0} l(V_{ij}, W_i, H_j) \), where \( l(V_{ij}, W_i, H_j) = (V_{ij} - W_i H_j)^2 \). We denote a subset of the nonzero entries in \( V \) as training set \( Z \). With a step size \( \epsilon \), an SGD algorithm for MF can be described in Alg. 2 \(^1\) ([60, 93]). Alg. 2 describes a serial algorithm that is not bound to a particular system. Convergence of the algorithm is measured by a training loss defined over the training set \( Z \subseteq V \), i.e., \( L_{tr} = \sum_{i,j: Z_{ij} \in Z} l(Z_{ij}, W_i, H_j)^2 \).

**Algorithm 2: SGD For Matrix Factorization**

**Input**: the training set \( Z \) and rank \( r \)
**Output**: factor matrices \( W \) and \( H \)
Randomly Initialize \( W \) and \( H \)
while not converged do
  for \( Z_{ij} \in Z \) do
    \[
    W_i \leftarrow W_i - W_i \epsilon \frac{\partial}{\partial W_i} l(Z_{ij}, W_i, H_j)
    \]
    \[
    H_j \leftarrow H_j - H_j \epsilon \frac{\partial}{\partial H_j} l(Z_{ij}, W_i, H_j)
    \]
  end for
end while

Similar to other iterative convergent ML algorithms, the heavy computation in SGD MF resides in the for-loop that iterates over the training set \( Z \). This is the work that should be parallelized for parallel training. Implementations of SGD MF on parameter server systems [44, 157] and graph processing systems [31, 64, 163] are often parallelized using data parallelism, where the training set \( Z \) is randomly partitioned and assigned to workers. Random partitioning leads to conflicting accesses on \( W \) or \( H \) and violating data dependence, e.g., if data samples \( Z_{ip} \) and \( Z_{iq} \), both read and write \( W_i \), but are assigned to different workers and executed in parallel at the same time.

\(^1\) Practical applications may employ regularization. Here we omit regularization for simplicity since it does not affect parallelization.
Serializable Parallelization of SGD MF

Pairs of data samples $Z_{ij}$ and $Z_{i'j'}$, $\forall i, j, i', j'$ : $i \neq i', j \neq j'$, are independent. That is, processing $Z_{ij}$ and $Z_{i'j'}$ does not read or write to the same entries in $W$ or $H$. Accordingly, we can devise a serializable parallelization by processing only independent data samples in parallel. Although different orderings of data samples may indeed lead to different numerical values of $W$ and $H$, serializability is sufficient for matching sequential execution’s convergence rate and model quality. Based on these observations, Gemulla et al. [60] proposed a serializable parallel SGD algorithm called stratified SGD.

Given a cluster of $P$ workers, Gemmulla et al.’s stratified SGD algorithm partitions the rating data matrix $Z$ into $P \times P$ blocks, and each block is denoted by its partition index $(i, j)$. The set of blocks is divided into $P$ strata, and the algorithm ensures that for any pair of blocks $(i, j)$ and $(i', j')$ in the same stratum, $i \neq i'$ and $j \neq j'$. In stratified SGD, an epoch (i.e., a full data pass) is divided into $P$ sub-epochs, where each sub-epoch processes one stratum and the $P$ blocks within a stratum are processed in parallel by $P$ workers. Processing rating matrix block $(i, j)$ reads and writes the $i$-th and $j$-th blocks of matrix $W$ and $H$, respectively. Stratified SGD enforces synchronization between workers at the end of each sub-epoch, and thus its parallel computation schedule ensures serializability as different workers do not access the same parameters within each sub-epoch, and a worker is guaranteed to observe the updates made by other workers in previous sub-epochs.

4.2.3 Communicating Model Parameters

Between-sub-epoch synchronization and parameter communication can be achieved using different strategies. Two of these strategies are efficiently supported by Spark, and two are not. We discuss these strategies in this section. Without losing generality, for the discussion in this section, we assume $H$ is smaller in size. To avoid communicating the larger matrix $W$, we assign to the $p$-th worker the rating data blocks $(p, 1), (p, 2), \ldots, (p, P)$ as well as the $p$-th block of $W$. In our Spark implementation, this is achieved by joining the RDD of the partitioned rating data $Z$ and the RDD of matrix $W$.

Communication Strategies Supported By Spark

Broadcasting. We can let the Spark driver program construct the $H$ matrix as a local variable and then broadcast it to workers. At the end of a sub-epoch, the driver program may retrieve the updated values of $H$ by invoking the collect operation on the RDD that contains copies of $H$. They can then be broadcasted for the next sub-epoch. While simple and straightforward, this approach fails to scale to a large number of parameters or workers. The driver node needs to be able to hold at least one copy of the $H$ matrix, which may contain billions of parameters in a modest problem. Moreover, the driver program sends a full copy of the $H$ matrix to each worker even though they only need $1/P$ of $H$, wasting a substantial amount of network bandwidth. Since we seek a solution that can potentially scale to a large number of model parameters without having to be restricted by a single machine,
broadcasting is rejected.

**RDD Join.** The $H$ matrix can be stored as an RDD, so it’s not restricted by the size of any single machine. The $H$ RDD can be joined with the RDD that contains the partitioned $Z$ and $W$ matrices to have the $H$ values available for processing corresponding $Z$ blocks. Since a worker uses a different partition of $H$ in each sub-epoch, one join operation is needed per sub-epoch. Recall that creating a $P$-way parallelism requires partitioning the rating matrix into $P \times P$ blocks and takes $P$ sub-epochs for a full pass of the training data. Thus higher degree of parallelism (utilizing more workers) causes more joins. Joining two RDDs typically involves shuffling.

**Communication Strategies Not Supported By Spark**

**Pipelining.** As discussed above, the parallel computation schedule assigns blocks of $Z$ that have the same horizontal coordinates to the same worker and schedules different workers to process different vertical blocks in the same sub-epoch. By choosing a proper processing order of blocks for each worker, we ensure that across all sub-epochs, each worker only receives parameters (blocks of $H$) from another fixed worker and sends updated parameters to another fixed worker. Thus each worker directly sends its updated parameters to a statically designated successor. Partitioning a $Z$ block vertically into smaller blocks allows communication of parameters to begin before the computation of a block finishes and thus effectively overlap communication time with computation time. Compared to RDD join, pipelining avoids the overhead to dynamically determining the destination of parameters and, more importantly, hides some of the computation time. This computation and communication schedule is not supported by Spark, so we compare to a standalone implementation.

**Distributed Shared Memory.** The matrix $H$ could be stored and served by distributed shared memory, such as used in Parameter Servers [100, 157]. While this approach is similar to broadcasting, it does not suffer the scalability and redundant communication issues as experienced by Spark broadcasting. Firstly, the parameter server could be distributed, so it is not restricted by the capability of a single node. Secondly, each worker only has to fetch the parameters needed for the next sub-epoch instead of the whole $H$ matrix. We compare to a Parameter Server implementation called Bösen.

**4.2.4 Evaluation and Results**

In order to understand Spark’s performance with proper context, we compare the following implementations:

**Python-Serial** or simply **Serial**: a serial implementation in Python (v3.4.0) of the Stochastic Gradient Descent algorithm with Gemulla et al.’s rating matrix partitioning strategy [60]. It stores the rating matrix and parameter matrices on disk, loads blocks into memory when they are needed and writes them back to disk before loading data for the next sub-epoch.
**Python-Spark** or simply **Spark** is implemented on PySpark (v1.6.0) and uses RDD joins for communicating the model parameters.

**Pipelined-Standalone** or simply **Standalone** implements the scheduled pipelining communication strategy in C++ based on Gemulla et al.’s algorithm [60] using the POSIX socket interface for communication.

**Pipelined-MPI** or simply **MPI** implements the scheduled pipelining communication strategy in Python using MPI4Python (v1.3 and MPI version 1.4.3) for communication.

**Bösen** is a Parameter Server that provides a distributed shared memory implementation for storing model parameters. The Bösen implementation of SGD MF is parallelized using data parallelize. The rating data $Z$ is partitioned by row and thus parallel workers may incur conflicting parameter accesses on $H$. Moreover, the Bösen implementation performs one global synchronization per epoch, i.e., a worker does not propagate its parameter updates to other workers until its entire local data partition has been processed. Note that this is different from the implementation of the distributed shared memory strategy mentioned in Section 4.2.3.

**STRADS** supports the Scheduled Model Parallelism parallelization [90]. The static model-parallel SGD-MF implementation on STRADS is effectively the same partitioning and scheduling strategy as Gemulla et al.’s algorithm [60].

As we are mostly interested in the processing throughput of various implementations, we report time per data pass as our performance metric for comparison, which measures the time taken to process all data samples in the training set once. It should be noted the Bösen and STRADS implementations use the adaptive gradient algorithm [109] for step size tuning which significantly improves the per-data-pass convergence rate with slightly higher computational cost. Moreover, the Bösen implementation allows conflicting writes and employs staleness, which harms the per-data-pass convergence rate. The Spark, Python-Serial, Standalone and MPI implementations have the same per-data-pass convergence rate.

Our experiments used the datasets and configurations as shown in Table 4.1. Duplicated Netflix datasets are used for weak scaling experiments. Experiments are conducted using the PRObE Nome cluster [62], where each node contains 16 physical cores and 32GB of memory and nodes are connected via 10Gbps Ethernet.

**Single-Threaded Baseline**

Fig. 4.4a and 4.4b show the time per data pass with various implementations running on a single execution thread on a single machine on two smaller datasets: MovieLens10M and MovieLens. While all are implemented in Python, the serial and MPI implementations are 3 - 4× faster than the Spark implementation. The standalone C++ implementation is near or more than 2 orders of magnitude faster than the PySpark implementation.
<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th># Ratings</th>
<th># Users</th>
<th># Movies</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>MovieLens10M</td>
<td>134MB</td>
<td>10M</td>
<td>71.5K</td>
<td>10.7K</td>
<td>100</td>
</tr>
<tr>
<td>MovieLens</td>
<td>335MB</td>
<td>22M</td>
<td>240K</td>
<td>33K</td>
<td>500</td>
</tr>
<tr>
<td>Netflix</td>
<td>1.3GB</td>
<td>100M</td>
<td>480K</td>
<td>18K</td>
<td>500</td>
</tr>
<tr>
<td>Netflix4</td>
<td>5.2GB</td>
<td>400M</td>
<td>1.92M</td>
<td>72K</td>
<td>500</td>
</tr>
<tr>
<td>Netflix16</td>
<td>20.8GB</td>
<td>1.6B</td>
<td>7.68M</td>
<td>288K</td>
<td>500</td>
</tr>
<tr>
<td>Netflix64</td>
<td>83.2GB</td>
<td>6.4B</td>
<td>30.7M</td>
<td>1.15M</td>
<td>500</td>
</tr>
<tr>
<td>Netflix256</td>
<td>332.8GB</td>
<td>25.6B</td>
<td>122.9M</td>
<td>4.6M</td>
<td>500</td>
</tr>
</tbody>
</table>

Table 4.1: Datasets used for the experiments.

Figure 4.4: Single-threaded baseline

(a) MovieLens10M  
(b) MovieLens

Figure 4.5: Spark running on a single machine

(a) Spark overhead with increasing number of strata (i.e., number of sub-epochs)  
(b) Spark SGD-MF fails to speed up using more cores on the same machine

Overhead with Higher Parallelism

Gemulla et al.’s algorithm partitions the ratings matrix into $P$ strata, which each consist of $P$ blocks that can be processed in parallel. Thus $P$ represents the number of processors that can be effectively used and thus indicates the degree of parallelism. Ideally, we expect the time per epoch to be inversely proportional to the number of strata. However, since having more strata (i.e., more sub-epochs) incurs overhead, such as more synchronization barriers, the time per data pass may not decrease linearly as higher parallelism is introduced.
We demonstrate how such overhead affects Spark’s per-data-pass processing time by processing the same ratings matrix (MovieLens10M) with increasing number of strata (i.e., number of sub-epochs) using only a single core. As shown in Fig. 4.5a, the per-data-pass processing time almost doubled when the number of strata is increased from 1 to 4, though the overhead increases much more slowly after that.

**Strong Scaling**

We evaluated strong scalability of different implementations using the Netflix dataset. The time per data pass of three implementations, including Spark, MPI and Standalone, using a single core is shown in Fig. 4.6b. Generally, we scale up the number of cores employed on a single shared-memory machine first before scaling out to multiple machines. In this case, the number of sub-epochs is the same as the number of cores. However, with Spark, we used 4 strata with 1 core and 12 strata with 2 cores on the same machine as these are the minimum number of strata that don’t cause Spark to crash due to memory failure\(^2\). Surprisingly, the Spark implementation fails to gain speedup from using more cores on a single machine, as shown in Fig. 4.5b. Thus we scale the Spark implementation using one core on each of multiple machines. Since a higher number of strata introduces additional overhead, as shown in Fig. 4.5a, we used the minimum number of strata that can effectively utilize the given number of cores and not cause Spark to crash. It should be noted that the standalone implementation is about $130\times$ faster and $14.9\times$ faster than the Spark and MPI implementation, respectively, when running on a single core.

As shown in Fig. 4.6a, the Spark implementation gains a $5.5\times$ speedup using 8 cores, 1 core on each of 8 machines, but gains no speedup from using more cores. The standalone implementation gains a $6.3\times$ speedup with 8 cores on the same machine, but only $9\times$ with 16 cores. The limited scalability of the standalone code on shared memory is largely due to higher number of cores incurring $10\times$ more cache misses, as shown in Fig. 4.8b. The standalone implementation gains a $80\times$ speedup with 256 cores spread over 16 machines, with each data pass taking 4 seconds. There’s no further speedup from using more cores and

\(^2\)Cause unknown.
most of the time is now spent on communication. The MPI implementation scales similarly to the standalone implementation.

Strong scalability with respect to the number of machines and a comparison with two general-purpose ML systems Bösen and STRADS is shown in Fig. 4.7a, and the time per data pass with different implementations running on a single machine (Spark uses only 1 core) is shown in Fig. 4.7b.

The Bösen implementation runs slower than the standalone implementation on a shared-memory multi-core machine as the Bösen client library employs locking for concurrency control and uses the adaptive gradient method [110] that incurs high computation overhead than plain SGD. While STRADS (also using adaptive gradient) scales better than the standalone implementation, the standalone implementation still faster with less resources, achieving about 4 seconds per data pass with 16 machines and STRADS achieves 6 seconds per data pass with 64 machines (no further speedup with more machines for both implementations).

**Weak Scaling**

We evaluated the weak scalability of the standalone implementation using the Netflix dataset duplicated a number of times proportional to the number of cores. 8 cores were assigned for the original Netflix data; 32 cores, i.e., 2 Nome nodes, were assigned for 4× Netflix data; etc. With weak scaling, we double work (training dataset size) when we double resources and hope for execution time per epoch to remain fixed, representing linear weak scaling. While the time per data pass remains roughly unchanged up to 16× duplicated data, with the number of cores proportionally increasing, it increased considerably for the 64× and 256× duplicated datasets. Even the most optimized standalone implementation cannot linearly scale to larger datasets.

We failed to run the Spark implementation on even 4× duplicated Netflix data due to consistent executor-lost failure during the first data pass. No hardware failure was observed.
4.2.5 Discussion

Spark’s ease of programming comes at a high performance degradation. The immutability of RDD makes it easy to parallelize the DAG computation and achieve fault tolerance. However, the rigid map-reduce execution model makes it impossible to implement the highly customized pipelining communication. The lack of random reads and in-place updates on RDDs restricts parameter updates to be performed by inefficient RDD joins or non-scalable broadcast.

The standalone pipelining implementation achieves best efficiency, i.e., achieving highest throughput using least amount of resources, but requires high programmer effort to implement inter-machine communication, worker coordination, etc. Machine learning frameworks such as STRADS and Bösen achieve high throughput and efficiency that’s close to the standalone implementation and substantially reduces programmer effort by abstracting away many system details, such as network communication, parameter caching, etc but still requires non-trivial manual parallelization effort, especially when implementing a parallel computation schedule that’s more complex than data parallelism.

4.3 Summary

Manually parallelized machine learning programs such as LightLDA on Bösen and SGD MF, including standalone implementation and implementations on STRADS and Bösen achieves high performance but require substantial programmer effort. Thanks to its higher level of abstraction, Spark indeed greatly simplifies application development compared to manual parallelization (a few days vs. a couple of weeks or even months), but the experimental evaluation suggests that Sparks suffers major performance penalty.

We desire a framework that provides an appropriate higher-level abstraction than STRADS’ and Bösen’s to simplify machine learning application development while achieves highly efficient execution. A good high-level abstraction captures the key characteristics of the appli-
cation computation and is represented by an intermediate representation that enables optimizations by system. Existing distributed computing systems, such as DryadLINQ [169] and Spark, successfully apply this approach and achieve high performance for analytical applications, but their programming abstraction is not well suited for machine learning, due to their restrictive execution model and the lack of mutable states.
Chapter 5

Scheduling Computation via Automatic Parallelization

While parallelization that preserves a serial algorithm’s data dependence improves the learning algorithm’s convergence rate, it requires substantial programmer effort to manually analyze data dependence and implement efficient distributed programs. Moreover, violating minor data dependence may increase the degree of parallelism with negligible impact on convergence rate. Such opportunities requires domain knowledge of the application program to explore. In this chapter, we present a holistic approach for automating dependence-aware parallelization and its implementation – Orion. Orion’s programming model and system abstraction natively support frequently mutated states and capture the access pattern on mutable states for parallelization. In this way, Orion substantially reduces programmer effort while achieving competitive performance compared to state-of-the-art manual parallelization.

5.1 Dependence-aware Parallelization

Given a training dataset \( \mathcal{D} = \{ \mathcal{D}_i | 1 \leq i \leq N \} \) where \( \mathcal{D}_i \) denotes a mini-batch of one or multiple data samples, a serial training algorithm computes an update function \( \Delta (A_t, \mathcal{D}_i) \) for each mini-batch \( \mathcal{D}_i \) using the current parameter values \( A_t \) and updates the parameters before processing the next mini-batch. Note that some ML algorithms update model parameters after processing each data sample, which is a special case that has a mini-batch size of 1. With an application-defined mini-batch size, our discussion is focused on the dependence across mini-batches.

Training algorithms typically take many passes (i.e., iterations) over the training dataset before they converge. In many ML applications, \( \Delta \) reads only a subset of the model parameters and generates refinements to a (possibly different) subset of parameters. If each worker is assigned with a mini-batch \( \mathcal{D}_k' \) such that the read-write sets of all \( \Delta (A_t, \mathcal{D}_k') \) computations are disjoint, then the parallel execution of multiple mini-batch computation is serializable. That is, the parallel execution produces the same result as a serial execution following some sequential ordering of the mini-batches. We refer to this style of paral-
Figure 5.1: Data parallelism vs. dependence-aware parallelism: (a) the read-write (R/W) sets of data mini-batches $D_1$ to $D_4$; (b) in data parallelism, mini-batches are randomly assigned to workers, leading to conflicting parameter accesses; (c) in dependence-aware parallelization (note that $D_4$ instead of $D_2$ is scheduled to run in parallel with $D_1$), mini-batches are carefully scheduled to avoid conflicting parameter accesses.

parallelization that preserves data dependence among mini-batches as dependence-aware parallelization. Fig. 5.1 compares data parallelism with dependence-aware parallelism. Note that under the dependence-aware parallelization shown, the parallel execution is equivalent to sequentially processing mini-batches $D_1$, $D_4$, $D_2$, and $D_3$ (serializable), while under the shown data-parallelism, execution is not serializable.

STRADS \[90\] is a scheduler framework for traditional model-parallel ML programs. It exploits independent parameter access from different data samples to achieve state-of-the-art convergence rate for SGD MF and topic modeling (LDA), which is considerably faster compared to data parallelism when such parallelization is applicable. However, STRADS requires programmers to manually parallelize the training algorithm, which demands significant programmer effort and is error-prone.

Generally, with manual parallelization, programmers identify the data dependences among loop iterations based on how they access shared memory and devise a computation schedule. A computation schedule breaks down the iteration space (e.g., $Z$) into partitions, which conceptually form a dependency graph. An ideal partitioning provides sufficient parallelism (i.e., many partitions can be executed in parallel) while amortizing synchronization overhead (i.e., partitions are large enough). The computation schedule also assigns partitions to workers. Dependencies among iteration space partitions incur synchronization among workers and network communication. Partition assignment affects synchronization frequency and communication volume.

In contrast, our system Orion automates dependence-aware parallelization of serial imperative ML programs for efficient distributed execution. Orion’s parallelization strategies are similar to STRADS but our focus is on automating dependence analysis and dependence-aware parallelization for serial imperative ML programs. Compared to Orion, STRADS performs neither static or dynamic analysis, nor code generation. Application programmers

---

1STRADS is open-sourced here: https://github.com/sailing-pmls/strads (last visited: 1/10/2019). SGD MF is not part of the open-sourced repository and was obtained from STRADS authors.
thus manually analyze data dependence and derive a computation schedule. While deriving an efficient computation schedule is most challenging, implementation is also highly non-trivial. SGD MF on STRADS is implemented as a coordinator and a worker program, totally consisting of 1788 lines of C++ code. The application program is responsible for coordination among workers, data partitioning, parameter communication and synchronization, etc. Due to STRADS’ low-level abstraction, there may be little code reuse across STRADS applications. STRADS-AP [91] simplifies application programming by performing a “virtual iteration” that does dynamic analysis for dependences that do not change in latter iterations.

While imperative programming with a shared memory abstraction is highly expressive and natural for programmers, parallelization is more difficult compared to functional programming as dependency has to be inferred from memory accesses. Orion employs static dependence analysis and parallelization techniques from automatic parallelizing compilers and takes advantage of ML-specific properties to relax program semantics and thus improve parallelism. Semantic relaxations include programmer-controlled dependence violation, which enables data parallelism with few code changes. Orion’s programming model abstracts away worker coordination by providing high-level primitives such as @parallel_for, map and groupBy. Moreover, Orion generates code for data loading, partitioning, and prefetching, tailored to specific data types, so that a computation schedule can be reused for different computation and data types without losing efficiency. Thus application programmers can focus on the core ML algorithm. Moreover, Orion minimizes remote random access overhead via automated data partitioning and bulk prefetching based on the memory access pattern discovered in static analysis to achieve efficient distributed execution.

Experiments on a number of ML applications confirm that preserving data dependence can significantly improve ML training’s convergence progress and our proposed techniques are effective. We also compare Orion with various offline ML training systems [16, 90, 157] and show that Orion achieves much better or matching convergence progress and at least comparable computation throughput, even when compared with state-of-the-art manual parallelization, while substantially reducing programmer effort.
5.2 Orion Programming Model

Orion consists of a distributed runtime and an application library (Fig 5.2). Orion application programmers implement an imperative **driver program** that executes instructions locally and in Orion’s distributed runtime using the application library. Distributed programming in Orion seamlessly integrates with the rest of the program thanks to Orion’s distributed shared memory (DSM) abstraction and parallel for-loops. Our prototype implementation supports application programs written in Julia [24]. Julia is a scripting language that offers high programmer productivity like Python with great execution speed [6] using just-in-time compilation.

5.2.1 Distributed Arrays

Orion’s main abstraction for DSM is a set of multi-dimensional matrices, which we refer to as Distributed Arrays (or DistArrays). A DistArray can contain elements of any serializable type and may be either dense or sparse. A DistArray is partitioned and stored in the memory of a set of distributed machines in Orion’s runtime and Orion automatically repartitions DistArrays to minimize remote access overhead when executing distributed parallel for-loops.

Elements of an \(N\)-dimensional DistArray are indexed with an \(N\)-tuple \((p_1, p_2, ..., p_n)\). A DistArray supports random access via both point queries (e.g., \(A[1, 3, 2]\)) to access a single element and set queries (e.g., \(A[1:3, 3, 2]\)) where a range is specified for one or multiple DistArray dimensions. Here \([1, 3, 2]\) and \([1:3, 3, 2]\) are DistArray subscripts, analogous to DSM addresses. Statements that access DistArray elements can either execute locally or in Orion’s distributed workers by using the parallel for-loop primitive.

Similar to Resilient Distributed Datasets (RDD) [174], DistArrays can be created by loading from text files using a user-defined parser or by transforming an existing DistArray using operations like map and groupBy. Text file loading and map operations are recorded by Orion and not evaluated until the driver program calls **materialize**. This allows Orion to fuse the user-defined functions across operations and avoids memory allocation for intermediate results. Unlike RDDs, set operations that may cause shuffling, such as groupBy, are evaluated eagerly for simplicity. We expect the performance impact of this simplification to be small for machine learning programs as the heavy computation happens in for-loops and is parallelized using parallel for loop (Sec. 5.2.2).

Compared to RDD, DistArray supports indexed random accesses (i.e., point and set queries) and in-place updates, which makes DistArray better suited for holding trainable model parameters which are iteratively updated, especially when each mini-batch updates only a subset of the parameters. A DistArray is automatically distributed among a set of worker machines and can be sparse. At the lowest level, TensorFlow tensors are dense matrices that reside on a single device and TensorFlow applications may manually represent sparse and distributed matrices using dense tensors (e.g., [136]). While DistArray does not provide a rich set of linear algebra operations like TensorFlow tensors, a DistArray set query
returns a Julia Array, which can leverage the rich set of linear algebra operations natively provided by Julia.

5.2.2 Distributed Parallel For-Loop

The driver program may iterate over the elements of an \( N \)-dimensional DistArray using a vanilla Julia for-loop. For example, the loop in Fig. 5.3 iterates over each element of DistArray \( A \) where \( e_{\text{idx}} \) is the element’s index and \( e_{\text{val}} \) is the element’s value. As \( A \) is an \( N \)-dimensional matrix, the for-loop is naturally an \( N \)-level perfectly nested loop and the DistArray represents the loop nest’s iteration space. Each DistArray element corresponds to a loop iteration and the element’s index \( e_{\text{idx}} \) is the loop iteration’s index vector, of which each element is referred to as a loop index variable.

For-loops iterating over a DistArray can be parallelized across a set of distributed workers using a \@parallel_for macro. Depending on the loop body’s access pattern to other DistArrays, parallelization assigns iterations to workers and adds synchronization when it is needed for preserving data dependence among loop iterations (i.e., loop-carried dependence). Iterations that have dependences between them because of shared accesses on DistArrays are executed one after another in the correct order. Thus the parallel execution is equivalent to a serial execution of the loop (serializable).

Tools like OpenMP [48] and MATLAB parfor [7] also provide parallel for-loop primitives, provided that the programmer asserts the for-loops have no dependency among its iterations. But Orion’s \@parallel_for macro can be applied to loops that have dependences among iterations, and preserves loop-carried dependences. Moreover, Orion’s parallel for-loop executes in a distributed cluster while existing tools only apply to single machines.

Let \( \mathcal{P} = \{(p_1, p_2, ..., p_n)|\forall i \in [1, n]: 0 \leq p_i < s_i\} \) represent the iteration space of a \( n \)-dimensional DistArray, where \((p_1, p_2, ..., p_n)\) represents the index vector of an iteration, and the size of the iteration space’s \( i \)-th dimension is \( s_i \). For any two iterations \( \vec{p} = (p_1, p_2, ..., p_n) \) and \( \vec{p}' = (p'_1, p'_2, ..., p'_n) \), Orion can parallelize the for-loop while preserving all loop-carried dependences if one of the following is true:

1. **1D Parallelization**: There exists a dimension \( i \) such that when \( p_i \neq p'_i \), there doesn’t exist any loop-carried dependence between iteration \( \vec{p} \) and iteration \( \vec{p}' \). Note that this also includes the case when there’s no dependence between any iterations.
2. **2D Parallelization**: There exist two dimensions \( i \) and \( j \) such that when \( p_i \neq p'_i \) and \( p_j \neq p'_j \), they both access the same DistArray element and at least one of the accesses is a write.
there doesn’t exist any loop-carried dependence between iteration \( \vec{p} \) and iteration \( \vec{p}' \).

3. **2D Parallelization w/ Unimodular Transformation**: When neither 1D nor 2D parallelization is applicable, in some cases (see Sec. 5.3.3), unimodular transformations [159] may be applied to transform the iteration space to enable 2D parallelization.

**Applicability.** Static parallelization requires the size of the iteration space to be constant and known at compile time. ML training applications usually iterate over a fixed data set or model parameters and Orion just-in-time compiles a for-loop after the iteration space DistArray is loaded or created. Orion’s dependence-aware parallelization strategies apply to for-loops when the loop body accesses only a subset of the shared memory addresses and the addresses can be fully determined given the loop index variables, i.e., the iteration-space DistArray index. More specifically, our current implementation accurately captures dependence when DistArray subscripts contain at most one loop index variable plus or minus a constant at each position. A more complex subscript is conservatively regarded as that it may take any value within the DistArray’s bounds. The loop body may inherit any driver program variable. The inherited variables are assumed to be read-only \(^3\) during a single loop execution but their values could change between different executions of the same loop.

ML applications commonly represent data records as a mapping from a \( n \)-tuple key to a value, i.e., \((k_1, k_2, \ldots, k_n) \rightarrow \text{value}\), where the key uniquely identifies the data record. Thus data records may be organized in a \( n \)-dimensional tensor, indexed by the key tuple. When parameter accesses are also indexed by the key tuple, parallelization via static dependence analysis is possible. For example, the popular bag-of-words model represents text as a set of mappings from a word to its number of occurrences. ML applications on text data often have parameters associated with each word, such as the word topic count vector in topic modeling with Latent Dirichlet Allocation or the word embedding vector, which are accessed based on word ID.

Deep neural network (DNN) training is an increasingly important class of ML workloads. The core computation of a typical DNN training program is a loop that iterates over data mini-batches where each iteration performs a forward pass, a backward pass and updates the neural network weights. DNNs commonly read and update all weights in each iteration, therefore serializable parallelization over mini-batches is not applicable. DNN training is most commonly parallelized with data parallelism, which can be achieved in Orion by permitting dependence violation as discussed in Sec. 5.2.3.

5.2.3 **Distributed Array Buffers**

Static dependence analysis avoids materializing a huge dependence graph whose size is proportional to the training dataset. Such a graph could be too expensive to store and an-

\(^3\)The loop body may still write to those variables but the new value is visible only to the worker that performs the write.
analyze. However, static dependence analysis requires the DistArray subscripts to be determined (as an expression of loop index variables and constants) statically to accurately capture the dependence among loop iterations.

First, some ML models, such as DNNs, perform dense parameter accesses. Second, while parameter accesses might be sparse in some models, the DistArray subscripts may depend on runtime values (e.g., $e_{val}$ in Fig. 5.3). For example, in sparse logistic regression, processing a data sample reads and updates the weights corresponding to the sample’s nonzero features. In this case, traditional dependence analysis conservatively marks all DistArray positions as accessed, leading to false dependences among iterations and impeding parallelization. For these models, serializable parallelization can be severely limited in computation throughput or simply inapplicable, therefore such ML training applications are often parallelized with dependence violations. The algorithm converges better (closer to serial execution) when there are fewer collisions and when writes make small changes. In order to support these applications, Orion application programmers may selectively exempt certain (or all) writes from dependence analysis using Distributed Array Buffers (or DistArray Buffers). By applying all writes to DistArray Buffers instead of DistArrays, an Orion application effectively resorts to data parallelism.

A DistArray Buffer is a write-back buffer of a DistArray, and provides the same API for point and set queries. A DistArray Buffer maintains a buffer instance on each worker, which is usually initialized empty. The application program may apply a subset of DistArray writes to a corresponding DistArray Buffer and exempt those writes from dependence analysis, making it possible to parallelize a for-loop that can’t be parallelized otherwise.

Typically the buffered writes are applied to the corresponding DistArray after the worker executes multiple for-loop iterations. The application program may optionally bound how long the writes can be buffered. Orion supports an element-wise user-defined function (UDF) for applying each DistArray Buffer’s buffered writes. This UDF is executed atomically on each DistArray element and thus supports atomic read-modify-writes. The UDF for applying buffered writes allows applications to define sophisticated custom logic for applying updates, and makes it easy to implement various adaptive gradient algorithms [54, 110, 142].

5.2.4 Putting Everything Together

Fig. 5.4 shows a Julia SGD MF program parallelized by Orion. The serial program has less than 90 lines of Julia code and can be parallelized by changing only a few lines. The parallel program creates DistArrays instead of local matrices for training data (ratings) and parameters ($W$ and $H$) by loading from text files (text_file) or random initialization (randomn). DistArrays can be manipulated with set operations, like map (e.g., line #9). The for-loops that iterate over the ratings matrix entries (e.g., line #14) are parallelized by applying the @parallel_for macro.

The parallel for-loop’s loop body may read any driver program variable that is visible
1 step_size = 0.01
2 # Omitted variable and function definitions
3 Orion.@dist_array ratings =
4 Orbit.text_file(data_path, parse_line)
5 Orion.materialize(ratings)
6 dim_x, dim_y = size(ratings)
7 Orion.@dist_array W = Orion.randn(K, dim_x)
8 Orion.@dist_array H = Orion.randn(K, dim_y)
9 Orion.@dist_array W = Orion.map(W, init_param, map_values=true)
10 Orion.@dist_array H = Orion.map(H, init_param, map_values=true)
11 Orion.materialize(W)
12 Orion.materialize(H)
13 Orion.@accumulator err = Float32(0.0)
14 for iter = 1:num_iterations
15     Orion.@parallel_for for (key, rv) in ratings
16         W_row = @view W[:, key[1]]
17         H_row = @view H[:, key[2]]
18         pred = dot(W_row, H_row)
19         W_grad .= -2 * (rv - pred) * H_row
20         H_grad .= -2 * (rv - pred) * W_row
21         W[:, key[1]] .= W_row - W_grad * step_size
22         H[:, key[2]] .= H_row - H_grad * step_size
23     end
24     Orion.@parallel_for for (key, rv) in ratings
25         pred = dot(W_row, H_row)
26         err += abs2(rv - pred)
27     end
28     err = Orion.get_aggregated_value(:err, :+)
29     Orion.reset_accumulator(:err)
30 end

Figure 5.4: SGD Matrix Factorization Parallelized using Orion
Input: a for-loop to parallelize

```
for (key, rv) in ratings
    W_row = @view W[:, key[1]]
    H_row = @view H[:, key[2]]
    pred = dot(W_row, H_row)
    diff = rv - pred
    W_grad .= -2 * diff * H_row
    H_grad .= -2 * diff * W_row
    W[:, key[1]] .= W_row - W_grad * step_size
    H[:, key[2]] .= H_row - H_grad * step_size
end
```

Figure 5.5: Overview of Orion’s static parallelization process using SGD MF as an example.

to the loop (e.g., step_size) and the driver program may access the result of a parallel for-loop execution by reading from DistArrays or by using an accumulator (e.g., err). When an accumulator variable is created (e.g., line #12), an instance of this variable is created on each Orion worker, and the state of each worker’s accumulators are retained across for-loop executions. The driver program may aggregate the value of all workers’ accumulators using a user-defined commutative and associative operator (e.g. line #25). The driver program may also execute arbitrary statements on workers, including defining local states.

5.3 Static Parallelization

Given Orion’s expressive programming model, in this section, we discuss how for-loops are parallelized and scheduled, along with various novel techniques to improve distributed execution throughput without programmer effort.

5.3.1 Parallelization Overview

Orion’s @parallel_for primitive is implemented as a Julia macro, which is expanded when the for-loop is compiled. A Julia macro is a function that is invoked during compilation (as opposed to at runtime), which takes in an abstract syntax tree (AST) and produces a new AST to be compiled by the Julia compiler. Orion’s @parallel_for macro statically analyzes the for-loop’s AST to compute dependences among loop iterations based on the loop body’s access pattern to DistArrays. These dependences are represented as dependence vectors.
∀ iterations \( \vec{p} = (p_1, p_2) \), \( \vec{p}' = (p'_1, p'_2) \) and
\( \vec{p}' > \vec{p} \), \( \vec{p}' \) depends on \( \vec{p} \) if
\( (\cdot, p_1) == (\cdot, p'_1) \) or \( (\cdot, p_2) == (\cdot, p'_2) \), i.e.,
\[
\begin{cases}
\because : & p_1 = p'_1 \\
\because : & p_2 = p'_2
\end{cases}
\]
The dependence vectors are thus \((0, \infty)\) and \((\infty, 0)\).

Since ∀ dependence vector \( d = (d_1, d_2) \), \( d_1 = 0 \)
or \( d_2 = 0 \), unordered 2D parallelization is
applicable. Partition ratings by its 1st and 2nd dimension.

\( W \) and \( H \) are range partitioned by its 2nd di-
mension and allocated among executors. The
smaller one of \( W \) and \( H \) is rotated among
executors.

\textbf{Figure 5.6:} Overview of Orion’s static parallelization process using SGD MF as an example.

Based on the dependence pattern, Orion decides whether the for-loop is 1D or 2D paral-
lelized and whether a unimodular transformation is needed (see Sec. 5.3.3). During macro
expansion, Orion generates functions that perform the loop body’s computation and de-
fines those functions in the distributed workers. According to the parallelization strategy,
the generated new AST, that executes on driver, invokes a static computation schedule with
the corresponding loop body functions. The generated AST also contains code that 1) repart-
titions relevant DistArrays to minimize remote access overhead; and 2) captures and broad-
casts driver program variables that are inherited in the loop body’s scope. Note that even
though the parallel for-loop may itself be inside of another for-loop and executed multiple
times, the macro expansion and compilation is executed only once. A global statement in a
Julia program is just-in-time compiled and executed before the following global statements
are compiled. Thus the compilation of a statement may make use of previous statements’
runtime execution results, such as DistArray sizes. Fig. 5.6 presents an overview of the JIT
compilation process using SGD MF as an example.

\section{Computing Dependence Vectors}

A lexicographically positive vector\(^4\) \( \vec{d} \) denotes a dependence vector of an \( n \)-loop nest if and
only if there exist two dependent iterations \( \vec{p}_1 \) and \( \vec{p}_2 \) such that \( \vec{p}_1 = \vec{p}_2 + \vec{d} \). Infinity \( \infty \)

\(^4\)A vector \( \vec{d} = (d_1, d_2, ..., d_n) \) is lexicographically positive if \( \exists i : d_i > 0 \) and \( \forall j < i : d_j \geq 0 \)
**Algorithm 3: Computing dependence vectors**

**input**: `refs` - the list of references on DistArray `D`  
**output**: `dvecs` - the set of dependence vectors due to references to `D`  

```plaintext
dvecs = EmptySet();
for each unique pair `ref_a` and `ref_b` in `refs` do
  ▷ Skip checking dependence if both references are read or if the loop is unordered and both references are write.
  if `(ref_a.is_read and ref_b.is_read) or (unordered_loop and ref_a.is_write and ref_b.is_write)` then
      continue;
  dvec = Vec(iter_space.num_dims, inf);
  independent = false;
  for `dim` ∈ `D.dims` do
    sub_a = ref_a.subs[dim];
    sub_b = ref_b.subs[dim];
    if `sub_a and sub_b contains a single loop index variable` then
      if `sub_a.dim_idx == sub_b.dim_idx` then
        dist = sub_a.const - sub_b.const;
        if `dvec[sub_a.dim_idx] != inf and dvec[sub_a.dim_idx] != dist` then
          independent = true;
          break;
        dvec[sub_a.dim_idx] = dist;
      else
        continue;
    else
      Test dependence for other subscript types;
  if not independent then
    correct dvec for lexicographical positiveness;
    dvecs = union(dvecs, {dvec});
```

69
(or positive/negative infinity, \( +\infty /-\infty \)) in dependence vectors means that the dependence vector may take any (positive or negative) integer value at that position. In Fig. 5.6, dependence vector \((0, \infty)\) means that any iteration \((p_1', p_2')\) depends on iteration \((p_1, p_2)\) as long as \(p_1' - p_1 == 0\). A dependence vector implies a dependence pattern shared by all iterations, yielding a concise dependence representation. However, dependence vectors may conservatively represent a dependence that exists for only certain iterations as a dependence for all iterations, unnecessarily limiting parallelism.

Many previous work discussed how to compute dependence vectors [88, 108]. An iteration depends on another (earlier) iteration if and only if they both access the same memory location and at least one of the accesses is a write. In general, computing dependence vectors requires performing a dependence test on the subscripts of each pair of DistArray references from two different iterations, and either prove independence or produce a dependence vector for the loop indices occurring in the scripts [88]. Since Orion currently supports accurate dependence capturing only for subscripts that contain at most one loop index variable plus or minus a constant at each position, we can simplify the algorithm. We represent each subscript as a 3-tuple \((\text{dim\_idx}, \text{const}, \text{stype})\), representing the loop index variable’s dimension index in the iteration space, the constant and the type of the subscript, i.e., whether it is a single value or a range and whether the subscript is supported for dependence analysis. Alg. 3 presents Orion’s core procedure for computing dependence vectors. Our algorithm produces at most one dependence vector from each pair of static DistArray references. Two DistArray references are independent when they are both read, and write-write dependence may be omitted when the loop iterations can be executed in any order (unordered\_loop). After skipping such reference pairs, we initialize a dependence vector whose elements are infinity, meaning that any two iterations may be dependent due to these two DistArray references. We then refine this conservative dependence by checking each subscript position. We declare the two references are independent if their subscripts will never match. In the end, we add the dependence vector to the set of dependence vectors after making sure it is lexicographically positive. The algorithm has a time complexity of \(O(N^2 \times D)\) for each referenced DistArray where \(N\) is the number of static DistArray references and \(D\) is the number of dimensions of the referenced DistArray.

5.3.3 Parallelization and Scheduling

Orion partitions the iteration space based on dependence vectors so that different partitions can be executed in parallel. Each worker is assigned a number of iteration space partitions and synchronizes at most once per partition.

Figs. 5.7 through 5.12 show different parallelization strategies for a \(4 \times 4\) iteration space, depending on the dependence pattern between iterations. Ellipses denote loop iterations and edges denote dependence between iterations. Note that representing the dependence in (a) requires only 1 dependence vector, namely \((0, 1)\), and representing the dependence in (b) and (c) requires only 2 dependence vectors, namely \((1, 0)\) and \((0, 1)\). Iterations of the same color are executed in parallel. Rectangles denote iteration space partitions. Workers are
denoted as w0, w1, etc. M and N denote the number of unique time-dimension (vertical) and space-dimension (horizontal) indices. I and J denote the length of the time and space dimensions and $m = M / I, n = N / J$. Although it’s not shown here, typically each worker is assigned with multiple space-dimension indices for better load balancing and multiple time-dimension indices for pipelined parallelism (Sec. 5.3.4).

Given the set of dependence vectors $\mathcal{D}$, if there exists a dimension $i$ such that $\forall \vec{d} = (d_1, d_2, ..., d_n) \in \mathcal{D}, d_i = 0$, then any two iterations $\vec{p} = (p_1, p_2, ..., p_n)$ and $\vec{p}' = (p'_1, p'_2, ..., p'_n)$ are independent as long as $p_i \neq p'_i$. Partitioning the iteration space by dimension $i$ ensures that any two iterations $\vec{p}$ and $\vec{p}'$ from two different partitions are independent. Thus the loop can be scheduled by assigning different iteration space partitions to different workers as there’s no data dependence across partitions. This is referred to as 1-dimensional (i.e. 1D) parallelization. Note that all such dimensions $i$ that satisfy the above condition are candidate partitioning dimensions. Fig. 5.7 shows an example that applies 1D parallelization to a 2-level loop nest and partitions the 2D iteration space by dimension $j$. The corresponding compute schedule is shown in Fig. 5.8. The workers synchronize with each other after executing all iterations in its assigned partition.

If there exist two dimensions $i$ and $j$ such that $\forall \vec{d} = (d_1, d_2, ..., d_n) \in \mathcal{D}, d_i = 0, d_j = 0$, then any two iterations $\vec{p} = (p_1, p_2, ..., p_n)$ and $\vec{p}' = (p'_1, p'_2, ..., p'_n)$ are independent as long as $p_i \neq p'_i$ and $p_j \neq p'_j$. In this case, the loop can be parallelized by partitioning the iteration space by dimensions $i$ and $j$, which we refer to as 2-dimensional (i.e. 2D) parallelization (see Fig. 5.9). The partitions are assigned to workers based on one of the dimensions, e.g. $j$ in this case, which we refer to as the space dimension and the other dimension is referred to as the time dimension. The computation is executed in a sequence of global time steps.


**Figure 5.9:** 2D parallelization.

\[
\begin{array}{c}
\text{for } \text{time\_step} = 0:(M + N - 2) \\
\quad \text{in\_parallel for } j = 0:(N-1) \\
\quad \quad i = \text{time\_step} - j \\
\quad \quad \text{if } i >= 0 \&\& i < N \\
\quad \quad \quad \text{for iter in partition}[j, i] \\
\quad \quad \quad \quad \text{execute\_iteration(iter)} \\
\quad \quad \quad \text{synchronize()} \\
\end{array}
\]

**Figure 5.10:** 2D computation schedule.

**Figure 5.11:** Unordered 2D parallel.

\[
\begin{array}{c}
\text{for } \text{time\_step} = 0:(M-1) \\
\quad \text{in\_parallel for } j = 0:(N-1) \\
\quad \quad i = (j + \text{time\_step}) \% N \\
\quad \quad \text{for iter in partition}[j, i] \\
\quad \quad \quad \text{execute\_iteration(iter)} \\
\quad \quad \text{synchronize()} \\
\end{array}
\]

**Figure 5.12:** Unordered 2D computation sched.
Within each time step, multiple workers may execute a local partition in parallel, where the partition’s time dimension index is derived from the time step number to ensure that all parallel partitions’ indices differ in both space and time dimensions. We observe that a partition depends on only two other iteration space partitions from the previous time step and one of them belongs to the same worker. Thus a worker waits for a signal from a single predecessor worker to begin the next time step instead of a global synchronization barrier.

**Relaxing the ordering constraints.** Automatic parallelizing compiler preserves the lexicographical ordering of loop iterations and thus dependences indicate the execution ordering of dependent loop iterations, such as shown in Fig. 5.9. With the ordering constraints, simultaneous execution of two iterations might not be possible even when they do not access the same memory location. For example, in Fig. 5.9, even though they do not access the same memory location, iteration \((3, 1)\) cannot be executed in parallel with \((0, 0)\) due to the ordering constraints enforced by iteration \((3, 0)\).

Many ML algorithms, such as Gibbs sampling, do not require a particular ordering in which data samples or mini-batches are processed. Other algorithms such as stochastic gradient descent usually randomly shuffle the dataset before or during training. For such ML algorithms, even though different iteration ordering may result in different numerical values and thus affect convergence process. However, to our best knowledge, enforcing a particular ordering, such as the lexicographical ordering, has not been shown to be beneficial while sacrificing parallelism. Therefore, Orion’s parallelization by default ensures only serializability but not the lexicographical ordering. Application may enforce ordering by using the ordered argument in `@parallel_for`. Relaxing the ordering constraints allows Orion to reorder iterations to maximize parallelism: Orion schedules workers to start from different indices along the time dimension to fully utilize all workers (Fig. 5.11 and Fig. 5.12).

**Unimodular transformation.** When neither 1D or 2D parallelization can be directly applied, Orion may apply unimodular transformations on the iteration space when the dependence vectors contain only numbers or positive infinity to enable 2D parallelization. Parallelizing for-loops using unimodular transformations was introduced by Wolf et. al [159]. The set of dependence vectors after unimodular transformation denoted as \(D'\) satisfy that \(\forall \vec{d} = (d_1, d_2, ..., d_n) \in D' : d_1 > 0\) (all dependences are carried by the outermost loop). With the transformed loop nest denoted as \(L_1, L_2, ...L_n\), there’s no dependence between iterations of the innermost loop nest \(L_2, L_3, ...L_n\) in the same outermost loop \(L_1\). Thus the for-loop can be parallelized by partitioning the transformed iteration space by the outermost dimension and any combination of the inner loop dimensions. By reversing the transformation, we can derive a 2D partitioning of the original iteration space.

As multiple candidate partitioning dimensions may exist, Orion uses a simple heuristic to choose the partitioning dimension(s) among candidates that minimizes the number of DistArray elements needed to be communicated among Orion workers during loop execution. This heuristic can be overridden by the application program.
Figure 5.13: Pipelined computation of a 2D parallelized unordered loop on 4 workers. An ellipse represents a worker executing a partition (space_partition_id, time_partition_id). The workers access different partitions of DistArray D at different time steps. Partitions of D that are being used by workers are lime-colored and the partitions that are being communicated are pink-colored. At the beginning of the loop execution, each worker is assigned with 2 time partition indices and thus 2 partitions of DistArray D. Upon finishing the first time step, a worker sends out the updated D partition and immediately begins the next time step using its locally available D partition.

**Dealing with Skewed Data Distribution.** As the parallel for-loop’s iteration space is often sparse and the data distribution is often skewed, for example, when iterating over a skewed dataset, partitioning the iteration space into equal-sized partitions results in imbalanced workload among workers. Orion DistArrays support a randomize operation that randomizes a DistArray along one or multiple dimensions to achieve a more uniform data distribution. Furthermore, Orion computes a histogram along each partitioning dimension to approximate the data distribution, which is used to generate a more balanced partitioning.

**Fault tolerance.** An Orion driver program can checkpoint a DistArray by writing it to disk, which is eagerly evaluated. For ML training, a common approach is to checkpoint the parameter DistArrays every N data passes.

### 5.3.4 Reducing Remote Random Access Overhead

Generally, DistArray random access can be served by a parameter server. However, in this case, each random access potentially result in a remote access over the inter-machine network. The overhead of network communication is significant even when Orion workers cache DistArray values and buffer DistArray writes.

**Locality and pipelining.** Usually different workers read and write to disjoint subsets of elements of a DistArray. If the workers’ read/write sets are disjoint range partitions of a DistArray, the DistArray may be range partitioned among workers so random access to it can be served locally.

Under 2D parallelization, the DistArray range partition accessed by a worker may be different at different time steps and a worker has to wait to receive a DistArray partition from its predecessor before starting a new time step. When the ordering constraints can be
<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
<th>DSM</th>
<th>Programming Paradigm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataflow</td>
<td>Spark [174], DryadLINQ [169]</td>
<td>No</td>
<td>dataflow</td>
</tr>
<tr>
<td>Dataflow w/ mutable states</td>
<td>TensorFlow [16]</td>
<td>Yes</td>
<td>dataflow</td>
</tr>
<tr>
<td>Parameter Server</td>
<td>parameter server [100], Bösen [157]</td>
<td>Yes</td>
<td>imperative</td>
</tr>
<tr>
<td>PS w/ scheduling</td>
<td>STRADS [90]</td>
<td>Yes</td>
<td>imperative</td>
</tr>
<tr>
<td>Graph Processing</td>
<td>PowerGraph [64], PowerLyra [31]</td>
<td>Limited</td>
<td>vertex programming</td>
</tr>
<tr>
<td>Orion</td>
<td></td>
<td>Yes</td>
<td>imperative</td>
</tr>
</tbody>
</table>

Table 5.1: Comparing different systems for offline machine learning training.

relaxed (Fig. 5.12), Orion avoids the workers’ idle waiting time by creating multiple time-dimension partition indices per worker and letting the worker proceed to a locally available time-dimension partition index while waiting for data from its predecessor, as illustrated in Fig. 5.13.

**Bulk prefetching.** If the same elements of a DistArray are simultaneously accessed by different workers, for example, when it is updated by a DistArray Buffer, or the disjoint sets of elements cannot be obtained from efficiently partitioning the DistArray, the DistArray is served by a number of server processes, similar to a Parameter Server. In this case, in order to minimize the random remote access overhead, Orion prefetches DistArray reads in bulk.

In order to accurately determine which values to prefetch, existing Parameter Server systems rely on programmers to implement a “virtual iteration” besides the actual computation to provide the parameter access pattern [45] or to manually implement prefetching and cache management [100]. Orion automates bulk prefetching by synthesizing a function that generates the list of DistArray element indices that are read during the loop body computation. The generated function executes loop body statements that read from non-locally allocated DistArrays, but instead of reading DistArray elements and performing computation, those statements are transformed to only record the DistArray subscript value. Since the DistArray subscripts may depend on runtime values, such as loop index variable and driver program variables (which are captured and broadcasted to workers as read-only variables), the function also executes statements that the DistArray subscripts have a data or control dependence on with proper control flow and ordering. If a DistArray subscript depends on values read from DistArrays, computing it may incur an expensive remote access. Therefore, DistArray subscripts that depend on other DistArray values are not recorded for bulk prefetching. The code generation algorithm is in spirit similar to dead code elimination.

### 5.4 Offline ML Training Systems: System Abstraction and API

In this section, we review and compare existing offline ML training systems (Table 5.1) with Orion, with an emphasis on their programming model and parallelization strategy. We focus on dataflow systems and graph processing systems, which present two distinct programming models.
5.4.1 Batch Dataflow Systems and TensorFlow

Many systems [16, 120, 169, 174] adopt a dataflow execution model, where the application program constructs a directed acyclic graph (DAG) that describes the computation and the computation DAG is lazily evaluated only when certain output is requested. A popular system among them is Spark [174], in which each node of the DAG represents a set of data records called a Resilient Distributed Dataset (RDD) and the edges represent transformation operations that transform one RDD to another. A fundamental limitation of traditional dataflow systems is that their computation DAG does not allow mutable states in order to ensure deterministic execution, which makes updating model parameters an expensive operation. For example, mutable states in Spark such as driver local variables or accumulators, are not represented in the computation graph and are stored and updated by a single driver process. SparkNet [118] represents model weights as driver program local variables, which are broadcasted to workers to compute new weights. The new weights produced by workers are collected and averaged by the driver. Each broadcast and collection takes about 20 seconds.

TensorFlow [16] is a deep learning system which also adopts the dataflow programming model, where nodes of the computation DAG represent operations whose inputs and outputs are tensors flowing along the edges. TensorFlow introduces mutable states such as variable and queue into the computation graph to efficiently handle model parameter updates. A typical TensorFlow program constructs a DAG that implements the update operation processing a single mini-batch of data, where trainable model parameters are represented as variables. One approach to represent different mini-batch’s or data sample’s access pattern on individual model parameters is represent each mini-batch (or data sample) and model parameter as separate nodes in the DAG (i.e., statically unroll the whole loop), resulting in a huge DAG that’s expensive to store and analyze.

Alternatively, the computation can be described as a while-loop [170] iterating over mini-batches or data samples. A TensorFlow application may parallelize a while-loop by assigning different operations of the loop body to different computing devices, and different devices may compute operations from different iterations. While TensorFlow while-loop allows different iterations to be executed in parallel, each operation is still assigned with and bound to a single computing device. In other words, TensorFlow’s while loop does not partition its iteration space among distributed devices and may fail to exploit the full parallelism enabled by the loop. On the other hand, TensorFlow while-loop enables additional parallelism for loops with a large and complex loop body (e.g., a multi-layer RNN), since the loop body can be distributed among multiple computing devices. Moreover, TensorFlow while-loop dynamically computes loop termination condition and supports data-dependent control flow inside the loop body including nested loops.
5.4.2 Graph Processing Systems

Graph processing systems [31, 64, 105, 106, 163, 176, 178] take a user-provided data graph as input and execute a vertex program on each graph vertex. Since a vertex program is restricted to access only data stored on that vertex itself, its edges or its neighboring vertices, the graph naturally describes the vertex program’s data dependence on mutable states. This property allows some systems to schedule independent vertex computation and ensure serializability by using graph coloring or pessimistic concurrency control [64, 105, 106]. However, graph coloring is an NP-complete problem and is expensive to perform; and with pessimistic concurrency control, lock contention may heavily limit the system’s scalability as demonstrated by a weak scaling experiment on PowerGraph [64]. As a result, recent graph processing systems have given up serializability: their vertex program either executes asynchronously or synchronizes with Bulk Synchronous Parallel synchronization [31, 163, 176, 178], both violating dependence among vertices.

5.5 Experimental Evaluation

Orion is implemented in $\sim 17,000$ lines of C++ and $\sim 6,300$ lines of Julia (v0.6.2) and has been open sourced.\(^5\) In this section, we evaluate Orion, focusing on parallelization effectiveness and execution efficiency. Our experiments were conducted on a 42-node cluster where each machine contains an Intel E5-2698Bv3 Xeon CPU and 64GiB of memory. Each CPU contains 16 cores with hyper-threading. These machines are connected with 40Gbps Ethernet.

5.5.1 Evaluation Setup and Methodology

We are interested in answering the following questions through experimental evaluation:

1. Is the training algorithms’ convergence rate sensitive to data dependence? Can dependence violation (such as data parallelism) significantly slow down algorithm convergence? Previous work (e.g., STRADS [90]) demonstrated that data dependence may have critical impact on algorithmic convergence and our results confirm their observations.

2. Can proper semantic relaxations such as relaxing the loop ordering constraints and violating non-critical dependences indeed improve computation throughput without jeopardizing convergence?

3. While preserving critical dependences, can Orion parallelization effectively speed up the computation throughput and thus overall convergence rate of serial Julia ML programs?

4. Do Orion applications achieve higher or competitive computation throughput and convergence rate compared to applications on other state-of-the-art offline ML training systems, including both manually parallelized data- and model-parallel programs?

\(^5\)URL: https://github.com/jinliangwei/orion
**ML applications.** We’ve implemented a number of ML applications on Orion, exercising different parallelization strategies, as summarized in Table 5.2. In this section, we focus on evaluating performance for SGD MF (w/o and w/ AdaRev) and LDA, which are commonly used benchmark applications and allow us to compare Orion with other systems.

**Datasets.** We evaluated SGD MF (w/o and w/ AdaRev) on the Netflix dataset [1] for movie recommendation, which contains \(\sim 100\) million movie ratings (rank is set to 1000). We evaluated LDA on a smaller NYTimes dataset that contains \(\sim 300\) thousand documents and a subset of the large ClueWeb dataset [2] that contains \(\sim 25\) million documents (32GB) (number of topics is set to 1000 and 400 respectively).

**Metrics.** Ultimately ML training applications desire to reach a high model quality in the least amount of time, which we refer to as *overall convergence rate*. A high overall convergence rate requires the training system to both process a large number of data samples per second, i.e., achieve a high *computation throughput*, and improve the model quality by a large margin per data pass, i.e., achieve a high *per-iteration convergence rate*. A serial execution typically achieves the best per-iteration convergence rate and thus serves as a golden standard. Different parallelizations may have different per-iteration convergence rate depending on whether and which data dependences are violated. Our evaluation metrics include both overall and per-iteration convergence rate to properly attribute the performance difference.

**ML systems in comparison.** We compared Orion with a number of state-of-the-art ML offline training systems on SGD MF (w/ and w/o AdaRev) and LDA in terms of both computation throughput and overall convergence rate. The systems that we experimentally compare to include Bösén parameter server [157], STRADS and TensorFlow.

Tu\(X^2\) [163] is a recently proposed graph processing system, particularly optimized for ML training workloads. Tu\(X^2\) was reported to have over an order of magnitude faster per-iteration time on SGD MF compared to PowerGraph [64] and PowerLyra [31]. With a rank of 50, Tu\(X^2\) SGD MF \(^6\) takes \(\sim 0.7\) seconds to perform one data pass on the Netflix dataset [1] using 8 machines, each with two Intel Xeon E5-2650 CPUs (16 physical cores), 256GiB of memory, and a Mellanox ConnectX-3 InfiniBand NIC with 54Gbps bandwidth (all higher than ours except for slightly slower CPUs). In contrast, Orion SGD MF achieves a per-iteration time of \(\sim 1.4\) seconds on 8 machines with the same number of CPU cores.

On the other hand, with a carefully tuned mini-batch size, Tu\(X^2\) SGD MF reaches a

---

\(^6\)Tu\(X^2\) is not open sourced

---

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Model</th>
<th>Learning Algorithm</th>
<th>LoC</th>
<th>Parallelizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGD MF</td>
<td>Matrix Factorization</td>
<td>SGD</td>
<td>87</td>
<td>2D Unordered</td>
</tr>
<tr>
<td>SGD MF AdaRev</td>
<td>Matrix Factorization</td>
<td>SGD w/ Adaptive Revision</td>
<td>108</td>
<td>2D Unordered</td>
</tr>
<tr>
<td>SLR</td>
<td>Sparse Logistic Regression</td>
<td>SGD</td>
<td>118</td>
<td>1D (data parallelism)</td>
</tr>
<tr>
<td>SLR AdaRev</td>
<td>Sparse Logistic Regression</td>
<td>SGD w/ Adaptive Revision</td>
<td>143</td>
<td>1D (data parallelism)</td>
</tr>
<tr>
<td>LDA</td>
<td>Latent Dirichlet Allocation</td>
<td>Collapsed Gibbs Sampling</td>
<td>398</td>
<td>2D Unordered, 1D</td>
</tr>
<tr>
<td>GBT</td>
<td>Gradient Boosted Tree</td>
<td>Gradient Boosting</td>
<td>695</td>
<td>1D</td>
</tr>
</tbody>
</table>

**Table 5.2:** ML applications parallelized by Orion.
nonzero squared loss (lower is better) of $\sim 7 \times 10^{10}$ in $\sim 600$ seconds using 32 machines in its best case, while Orion SGD MF reaches $\sim 8.3 \times 10^7$ in $\sim 68$ seconds using only 8 machines. Even though TuX$^2$ SGD MF achieves a higher computation throughput, its overall convergence rate is much lower than Orion's due to violating data dependence.

5.5.2 Summary of Evaluation Results

1. Preserving data dependence is critical for SGD MF (w/o and w/ AdaRev) and LDA. Dependence-violating parallelization (i.e., data parallelism) takes many more data passes than serial execution to reach the same model quality, while dependence-aware parallelization (even with proper semantic relaxations) retains a comparable per-iteration convergence rate to serial execution.

2. Orion-parallelized SGD MF (w/ and w/o AdaRev) and LDA converge significantly faster than manual data-parallel implementations on Bösen, in terms of both number of iterations and wall clock time.

3. Data-parallel SGD MF AdaRev and LDA on Bösen converges faster with more frequent communication of parameter values and updates, approaching Orion parallelization at the cost of higher network bandwidth.

4. Orion-parallelized SGD MF AdaRev and LDA achieve a matching per-iteration convergence rate to manual model-parallel programs on STRADS, but may have a slower time per iteration mainly due to Julia's language overhead compared to C++.

5. Orion-parallelized SGD MF converges considerably faster than a data-parallel implementation on TensorFlow while achieving a $2.2 \times$ faster per-iteration time.

5.5.3 Parallelization Effectiveness

<table>
<thead>
<tr>
<th>Application</th>
<th>Ordered</th>
<th>Unordered</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGD MF (Netflix)</td>
<td>13.1</td>
<td>5.9</td>
<td>2.2×</td>
</tr>
<tr>
<td>SGD MF AdaRev (Netflix)</td>
<td>43.6</td>
<td>16.7</td>
<td>2.6×</td>
</tr>
<tr>
<td>LDA (NYTimes)</td>
<td>29.9</td>
<td>5.0</td>
<td>6.0×</td>
</tr>
</tbody>
</table>

Table 5.3: Time per iteration (seconds) with ordered and unordered 2D parallelization (12 machines), averaged over iteration 2 to 100.

We compare Orion-parallelized Julia programs with serial Julia programs in terms of both computation throughput (i.e., time per iteration) and per-iteration convergence rate (Fig. 5.16). As shown in Fig. 5.14, although Orion abstraction incurs some overhead, Orion parallelization outperforms the serial Julia programs using only two workers and enables consistent speedup up to 384 workers. Although Orion’s parallelization relaxes the loop ordering constraints for both SGD MF and LDA, and violates some non-critical dependences in LDA, preserving (critical) dependences enable Orion parallelization to achieve a matching convergence rate to serial execution (Fig. 5.16a and Fig. 5.16b). On the other hand, data parallelism (using Bösen) converges substantially slower than serial execution due to violating all dependences.
Figure 5.14: Time (seconds) per iteration

Figure 5.15: Orion parallelization effectiveness: comparing the time per iteration (averaged over iteration 2 to 8) of serial Julia programs with Orion-parallelized programs. The Orion-parallelized programs are executed using different number of workers (virtual cores) on up to 12 machines, with up to 32 workers per machine.

Table 5.3 compares ordered and unordered 2D parallelization in terms of computation throughput. Theoretically, relaxing the loop ordering constraints at most doubles parallelism. But thanks to the more efficient communication scheme enabled by this relaxation (see section 5.3.4), which hides the communication latency, we observe a over $2 \times$ speedup. Fig. 5.16a and Fig. 5.16b show that loop ordering makes negligible differences in convergence rate. While we observe a bigger difference when adaptive revision [110] is used, relaxing the loop constraints is still beneficial for the improved computation throughput.

**Bulk Prefetching.** When training SLR using SGD, each data sample reads and updates a number of weight values corresponding to the nonzero features of the data record, which is unknown until the data sample is processed. The sequence of DistArray reads causes a sequence of inter-process communication, possibly over inter-machine networks. In a single-machine experiment using the KDD2010 (Algebra) [58] dataset, each data pass takes 7682 seconds, wasting most of the time on communication. Orion automatically synthesizes a function to prefetch the needed DistArray values in bulk (see Section 5.3.4) and thus reduces the per-iteration time to 9.2 seconds. It can be further reduced to 6.3 seconds by caching the prefetch indices.

### 5.5.4 Comparison with Other Systems

**Manual data parallelism.** Under data parallelism, Bösen workers synchronize after processing the entire local data partition. While achieving a high computation throughput,
data-parallel applications on Bösen converge considerablly slower than Orion-parallelized programs.

**Data parallelism w/ communication management.** Bösen features a communication management (CM) mechanism that improves the convergence rate of data-parallel training. Given a bandwidth budget, CM proactively communicates parameter updates and fresh parameter values before the synchronization barrier, when spare network bandwidth is available, to reduce the error due to violating data dependence. Moreover, CM prioritizes large updates to more effectively utilize the limited bandwidth budget. We assign each Bösen machine a bandwidth budget of 1600Mbps and 2560Mbps respectively for SGD MF and LDA for maximal overall convergence rate. For SGD MF on Netflix and LDA on ClueWeb25M, CM achieves similar per-iteration convergence rate compared to dependence-aware parallelization by Orion but is still $\sim40\%$ slower for LDA on NYTimes. For both SGD MF and LDA, CM uses substantially higher network bandwidth
than Orion due to the aggressive communication (Fig. 5.17) Excessive communication incurs CPU overhead due to marshalling and lock contention, reducing Bösen’s computation throughput and leading to a slower overall convergence rate than Orion when training LDA on ClueWeb25M.

**Manual model parallelism.** Compared to manually optimized model-parallel programs on STRADS, Orion-parallelized SGD MF AdaRev and LDA achieve a matching per-iteration convergence rate (Fig. 5.19). While achieving a similar computation throughput on SGD MF AdaRev, Orion takes $\sim 1.8 \times$ (ClueWeb25M) and $\sim 4.0 \times$ (NYTimes) longer than STRADS to execute an iteration for LDA. STRADS’s better performance is largely due to a communication optimization: communicating data between workers on the same machine requires only pointer swapping. Since Julia (v0.6.2) doesn’t yet support shared-memory multithreading, inter-process communication in Orion incurs marshalling and memory copies. This overhead is negligible for SGD MF where the communication is mostly float arrays with trivial serialization.

**TensorFlow.** We compare Orion-parallelized SGD MF with an implementation on TensorFlow (v1.8), both executed on a single machine using CPU (Fig. 5.20). Following TensorFlow (TF) common practices, our SGD MF program constructs a DAG which processes a mini-batch of data matrix entries to exploit TF's highly parallelized operators. Since TF does not update model parameters until a full mini-batch is processed, TF SGD MF converges considerably slower than Orion's iteration-wise. With a mini-batch size of 25 million, TF is $\sim 2.2 \times$ slower in terms of per-iteration time, partly due to redundant computation with respect to sparse data matrix (TF runs out of memory with larger mini-batch sizes). Each iteration takes longer with a smaller mini-batch size (Fig. 5.20b) because of not fully utilizing all CPU cores. Overall TF SGD MF converges much slower than Orion’s, indicating TF might not be the best option for sparse ML applications.

## 5.6 Related Work

**Automatic parallelizing compilers.** There has been decades of work on automatically parallelizing programs based on static data dependence analysis. This includes both vec-
Figure 5.18: Orion vs. B"osen, convergence on 12 machines (384 workers)
Figure 5.19: Orion vs. STRADS, convergence on 12 machines (384 workers)
orization [22, 122] and parallelization for multiple processors with a shared global memory, like Orion. Many loop transformation techniques have been developed for the latter, including loop interchange [160], loop skewing [161] and loop reversal. These transformations can be unified under unimodular transformations [159], which can only be applied to perfectly nested loops, e.g., traversing a multi-dimensional tensor. Affine scheduling [50, 56, 57] applies to arbitrary nestings of loops and unifies unimodular transformation with loop distribution, fusion, reindexing and scaling. Affine scheduling maps dynamic instances of instructions to a time space and instructions assigned the same time can be executed in parallel. Lim et al. [103] additionally partitions the instructions among processors to minimize synchronization.

**Dynamic analysis.** Pingali et al. [125] addresses parallelization by representing algorithms as operators and a topology, which describes the dependence between operators. The topology graph may be obtained from static analysis or dynamic tracing, or given as an input. Compared to static dependence analysis, this approach may be effective in parallelizing algorithms that deal with irregular data structures, e.g., graphs, but may suffer a larger overhead due to dynamic tracing and analyzing a large dependence graph.

**Approximate computing.** Previous work has proposed taking advantage of the approximate nature of application programs and introduced techniques, such as loop perforation [138] and task skipping [129] to reduce computation while sacrificing accuracy. Sampson et al. [134] rely on programmers to declare data that tolerates approximation so it can be mapped to lower-power hardware to save energy. HELIX-UP [28] also proposes to relax program semantics to increase parallelism and uses programmer-provided training inputs.
to tune the degree of approximation. Although auto-tuning could be incorporated in Orion, we believe that ML practitioners have domain-specific heuristics to make reasonable decisions while auto-tuning can be expensive.

5.7 Summary

We present Orion, a system that parallelizes ML programs based on static data dependence and unifies various parallelization strategies under a clean programming abstraction. Orion achieves better or competitive performance compared to state-of-the-art offline ML training systems while substantially reducing programmer effort. We believe that Orion is an effective first step towards applying static dependence analysis to parallelize imperative ML programs for distributed training.
Chapter 6

Scaling Model Capacity by Scheduling Memory Allocation

In previous chapters, we discussed scheduling network communication and computation to improve convergence time. Another precious resource is memory capacity: as discussed in Sec. 2.5.1, scaling model capacity improves accuracy, but model capacity is limited by memory size. In this chapter, we study how to schedule memory allocation across expensive GPU memory and cheap host memory to efficiently train larger models without incurring a high cost.

We present a number of techniques that leverage the large and cheap host memory to enable training larger DNNs on GPUs. While there have been a large number of previous works on reducing (GPU) memory consumption in training DNNs, we are the first to implement our techniques in a mature and popular deep learning system (i.e., TensorFlow) and demonstrate their effectiveness on a wide range of benchmarks instead of only convolutional neural networks. Moreover, our techniques support existing TensorFlow APIs with no modifications.

The heavy computational demand of DNNs motivates modern deep learning frameworks to use a dataflow graph as the intermediate representation of model computation. At runtime, the framework executes operations, i.e., vertices in the graph, following the data dependences, which are represented as edges. A key benefit of this informative representation is that it enables various global optimizations before execution, such as operator fusion and data layout transformation. Our techniques leverage TensorFlow’s dataflow graph to offload data from GPU to host memory, prefetch data to GPU, and makes best effort to avoid delaying computation.

We refer to our memory-optimized TensorFlow as TensorFlowMem. Compared to TensorFlow, TensorFlowMem partitions the computation graph, executes the graph partition by partition, and keeps only the data that’s relevant to the executing partition in GPU mem-

1Currently our techniques don’t support computation graphs that use dynamic control flow operators, though this extension should not require significant innovation
ory by offloading to and prefetching from host memory. Sequential execution across partitions allows TensorFlowMem to easily identify GPU tensors that are intermediate results between partitions, offload them to host memory, and prefetch them back to GPU memory shortly before they are used. While this is essentially paging, the computation graph allows TensorFlowMem to schedule data movements before the memory or the data value is needed to avoid slowing down computation.

We compared TensorFlowMem with vanilla TensorFlow on a broad range of neural network architectures, including CNN, RNN, Transformer, MoE, and GAN, as summarized in Table 6.2. We observed an up to 87% reduction in peak GPU memory consumption, with a runtime overhead (increase) of up to 3.4× (2.2× on average). The key benefit of TensorFlowMem is that it enables training much larger models without additional GPUs. For example, TensorFlowMem enables training a ResNet model of 1916 layers and a Mixture of Experts (MoE) model that has 2.5 billion parameters on a single GPU with 12 GB of memory, while vanilla TensorFlow fails to train ResNet models with more than 504 layers and MoE models with more than 0.66 billion parameters due to out-of-memory error. Furthermore, TensorFlowMem’s memory optimization techniques are applicable to distributed model-parallel training, making it possible to train even larger models on the same hardware. Additionally, TensorFlowMem also enables training using 3× to 4× larger mini-batch sizes than vanilla TensorFlow.

6.1 Related Work

One approach to training larger models is to distribute the model computation across many devices, referred to as model parallelism. GPipe [80] proposes to partition the neural network in a “layer-by-layer” fashion to enable training large neural networks on multiple GPUs or TPUs and uses pipeline parallelism across mini-batches to reduce communication overhead. PipeDream [71] additionally uses pipeline parallelism across mini-batches to hide the communication overhead. However, such an approach is limited by the number of layers a neural network has and is less helpful for networks that concentrate work in a few wide layers.

FlexFlow [86], Mesh-TensorFlow [136], and Tofu [154] propose to partition operations and place individual operations onto distributed devices. Although their fine-grained partitioning strategy may apply to models that fail to fit in memory using layer-wise partitioning, they do not take advantage of pipeline parallelism and communication remains a bottleneck. Therefore, they rely on high parallelism in the computation graph or individual operations to make full use of the compute power of all the GPUs.

For all these efforts, the model parallelism approach requires a large number of expensive GPUs to fit a large model. In this chapter, we exploit an alternative approach that leverages the relatively inexpensive host memory to enable training large models. Users can easily and inexpensively buy more CPU memory, but rarely use it for DL model parameters, activations or input batches because limitations of the computing framework. Our
approach is complementary to model-parallel distribution and, when combined, enables training even larger models. Moving data from/to host memory will slow computation, but larger models provide better accuracy, which is almost always more important than speed.

To avoid the distribution challenge, Chen et al. propose to use gradient checkpointing, which is a classic technique in automatic differentiation, in deep learning systems (i.e., MXNet) to recompute compute selected intermediate results to avoid storing them. GeePS [46] and vDNN [128] propose to leverage the layer-wise structure in DNNs (mostly convolutional neural networks) to offload and prefetch intermediate results (and parameters). While this approach is shown to be effective in Caffe and vDNN’s prototype system, it is difficult to directly apply to neural networks that do not have a strictly sequential, layer-wise structure and systems, such as TensorFlow, which employ a fine-grained graph representation that does not retain the layer-wise information. SuperNeurons [153] introduces a liveness analysis that extends the memory swapping mechanism to non-sequential neural network architectures, but is still implemented in a prototype system and is only evaluated on CNNs.

Meng et al. [113] applies the memory swapping mechanism in TensorFlow and their mechanism is similar to TensorFlow’s Grappler memory optimizer. They both rely on accurate estimations of operations’ execution time and memory usage. However, accurate estimation, as we show later in this chapter, is an unsolved problem and the Grappler memory optimizer fails to scale ResNet and Transformer to a larger size. The WhileLoop operation in TensorFlow supports memory swapping between loop iterations to leverage host memory to run loops that have many iterations, which appear in models, such as recurrent neural networks [170]. It does not support more general use cases. Salimans et al. [146] implemented a gradient checkpointing library for TensorFlow. Using it requires application programs to directly manipulate gradients, which makes it difficult, if not impossible to be used with high-level TensorFlow APIs, such as Estimators.

6.2 Background

6.2.1 Dataflow Graph As An Intermediate Representation For DNNs

Most deep learning systems represent the model computation as a graph. In older frameworks such as Caffe [85] and DistBelief [52], the models are composed of existing layers. For efficiency, the layers are implemented as C++ classes. While it is easy to build a model from existing layers, it’s difficult for advanced machine learning researchers to add new layers. The rigid computation pattern of these frameworks also makes it difficult to refine existing learning algorithms or define new learning algorithms.

In order to address the above limitations, modern frameworks such as TensorFlow [16] and Theano [21] represent computation as a dataflow graph of primitive mathematical operations. This representation makes it easy for users to define new layers and new training algorithms using primitive operations. The computation graph has also become much more complicated than a stack of layers. Besides stateless computation operations, the dataflow
graph also contains stateful operations such as Variable and Constant operations. Besides data inputs and outputs, the computation graph may also contain control dependency edges to enforce ordering among operations.

<table>
<thead>
<tr>
<th>Model</th>
<th>Key Feature</th>
<th>#Param.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformer</td>
<td>Attention</td>
<td>61 Million</td>
</tr>
<tr>
<td>Transformer w/ MoE on MeshTF</td>
<td>MoE</td>
<td>800 Million</td>
</tr>
<tr>
<td>ResNet-152</td>
<td>Convolution</td>
<td>60 Million</td>
</tr>
<tr>
<td>ResNet-1916</td>
<td>Convolution</td>
<td>697 Million</td>
</tr>
<tr>
<td>WGAN-GP (Gen. / Disc.)</td>
<td>Convolution</td>
<td>17 Million</td>
</tr>
<tr>
<td>Mozilla DeepSpeech</td>
<td>Recurrent</td>
<td>47 Million</td>
</tr>
</tbody>
</table>

**Table 6.1:** Deep Learning models (implemented on TensorFlow) used in our evaluation and the number of model parameters.

<table>
<thead>
<tr>
<th>Model</th>
<th>Depth</th>
<th>#Nodes</th>
<th>#Nodes / Depth</th>
<th>Avg. Indegree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformer</td>
<td>2731</td>
<td>7782</td>
<td>2.8</td>
<td>2.0</td>
</tr>
<tr>
<td>Transformer w/ MoE on MeshTF</td>
<td>2714</td>
<td>8392</td>
<td>3.1</td>
<td>1.8</td>
</tr>
<tr>
<td>ResNet-152</td>
<td>1054</td>
<td>5783</td>
<td>5.5</td>
<td>1.9</td>
</tr>
<tr>
<td>ResNet-1916</td>
<td>12814</td>
<td>68699</td>
<td>5.4</td>
<td>1.9</td>
</tr>
<tr>
<td>WGAN-GP (Gen. / Disc.)</td>
<td>2267 / 4093</td>
<td>5315 / 15112</td>
<td>2.3 / 3.7</td>
<td>2.0 / 2.0</td>
</tr>
<tr>
<td>Mozilla DeepSpeech</td>
<td>140</td>
<td>549</td>
<td>3.9</td>
<td>1.6</td>
</tr>
</tbody>
</table>

**Table 6.2:** Graph statistics for the DNN models used in benchmarks. Depth refers to the length of the longest path. The number of parameters in MoE is tunable and we report the smallest version that we used in our benchmarks here.

Table 6.1 presents a diverse set of DNN models that we use as benchmarks and Table 6.2 presents statistics of the TensorFlow computation graph of these models. Note that these graphs only include operations that are executed during training and do not include operations such as parameter initializations, etc. We observe that the graph are composed of thousands or even tens of thousands of operations (orders of magnitude more than the number of layers) and also have a long depth. Nodes in the graph have an average in-degree of nearly 2. These indicate that the computation graph is large and complex and is not a simple linear graph. For a linear computation graph, there exists one or a small number of valid topological ordering of the graph nodes, which makes it easy to predict the order in which the graph nodes are executed. However, generally speaking, the execution ordering for general dataflow graphs is hard to predict in TensorFlow.

6.2.2 TensorFlow

TensorFlow [16] is arguably one of the most widely used and mature deep learning systems. Since we implement and evaluate our techniques based on TensorFlow, we briefly review TensorFlow’s programming model, computation execution and memory management in this section. Fig. 6.1 presents an overview of TensorFlow’s execution of a computation graph to serve a query of an operation’s output.
Figure 6.1: TensorFlow Execution. Pattern indicates whether a node is a stateful (Variable or Constant) or stateless operation. Color indicates placement of the operation (CPU vs. GPU).
Programming Model. A TensorFlow application defines a computation, such as the forward pass of a neural network, as a directed graph, whose nodes represent operations. Edges between nodes represent output tensors that are fed into successor operations. While most operations (such as MatMul and Conv2D) are stateless functions, TensorFlow introduces a Variable operation to represent frequently updated model parameters. A Variable holds an internal buffer and outputs a reference handle to the buffer when executed. The reference handle allows other operations to read and update the Variable’s value in place. To simplify the implementation of SGD algorithm, TensorFlow supports automatic differentiation that generates back-propagation computation from the user-defined forward computation. TensorFlow also supports an application program explicitly declaring a placement constraint for each operation and thus allows a single computation graph to utilize a heterogeneous set of computing devices.

Graph Execution. After defining the computation graph, a TensorFlow application program may query an operation’s output, e.g., \( D \) in Fig. 6.1, which triggers graph execution. The requested value might depend on only a subset of operations in the computation graph and thus TensorFlow first creates an execution graph by pruning unnecessary operations. TensorFlow places operations of this execution graph onto computing devices, including GPUs, CPUs and TPUs, subject to the placement constraints specified by the application program. After placement, the execution graph is run through a series of optimization passes in Grappler, such as memory optimization, and constant folding. To facilitate computation across a set of distributed devices, TensorFlow adds a pair of Send and Receive operations that transmit tensors across devices. TensorFlow partitions the execution graph among a set of executors, each corresponding to a compute device.

An executor’s subgraph is executed in a breadth-first fashion, which begins with one of the executors executing a global SOURCE node which has no input dependency. When an executor finishes executing a node, it schedules to a thread pool all of its successors whose dependencies have been satisfied. A node’s successor could be a Send node that sends a control signal which triggers the execution of another executor’s subgraph. Note that when a node has a large fanout, the order in which its successor nodes are executed depends on the runtime schedule and can vary greatly, leading to challenges when aiming to limit their peak total memory usage.

Memory Management. TensorFlow allocates Variable (model parameters) and Constant nodes as persistent tensors, which holds memory until graph destruction. Unlike MXNet [34] and SuperNeurons [153], TensorFlow dynamically allocates memory for operations’ outputs during graph execution. When an operation generates an output tensor, a reference handle to the tensor is assigned to each successor operation that consumes the tensor. And the tensor’s reference count is incremented for each successor operation. After an operation is executed, TensorFlow decrements the reference count of its input tensors and frees a tensor when its reference count reaches zero. Dynamic memory allocation allows TensorFlow to reuse device memory for other operations. Like computation execution order, memory consumption during training depends on computation scheduling and is highly
variable.

**Graph Optimization.** TensorFlow’s Grappler module encompasses a number of optimization passes to improve execution speed and memory footprint of the computation graph. There are two Grappler optimizations that could considerably affect memory consumption and we briefly discuss them here. When a GPU’s peak memory consumption exceeds the GPU’s memory capacity during simulation, the Grappler memory optimizer performs a swapping pass that adds SwapOut and SwapIn nodes to offload GPU tensors to host memory and prefetch them to GPU before they are needed. This swapping pass makes the best effort to reduce peak memory consumption with minimal runtime overhead. However, in our experiments, we found that the swapping pass provides no benefit for scaling ResNet’s depth or MoE’s number of experts (for both single-GPU and distributed settings). TensorFlowMem also performs memory swapping, but possibly on different tensors and at different times, which may lead to conflicting swapping decisions. To avoid the complication of dealing with potential conflicts, TensorFlowMem supersedes Grappler’s swapping pass and we compare with TensorFlow with Grappler’s swapping pass disabled. Grappler also performs a constant folding optimization that folds a subgraph into a single Constant node that holds the result of the subgraph when the subgraph always evaluates to a constant. Constant folding reduces redundant computation but incurs higher memory consumption to store the previously evaluated result. In next section, we discuss TensorFlowMem’s memory optimizations.

### 6.3 Memory Optimizations for TensorFlow

In this section, we present memory optimizations for TensorFlow to reduce GPU memory consumption during training.

#### 6.3.1 A Motivating Example

![Figure 6.2: Mixture of Experts layer: example non-linear architecture.](image)

93
As mentioned before, the key idea of TensorFlowMem is partitioned execution and swapping intermediate results between GPU and CPU memory. Generally speaking, how a graph is partitioned is key to performance. When the graph is composed of a sequence of complex layers, it might be suitable to partition the computation graph layer by layer, which is the approach taken by GeePS [46] and vDNN [128]. However, in a general dataflow graph, it is difficult to apply this approach to a fine-grained graph, where the number of operations is much larger than the number of logical layers and operations are not associated with the logical layer information.

Even when such layer-wise grouping can be achieved or approximated, for example, by assigning operations of the same depth to the same graph partition, a single partition may produce too many intermediate results within the partition and exhaust the limited memory of the GPU devices or locate results across partitions incurring high communication overhead when accessed.

An example of such large complex layer is the Mixture of Experts (MoE) layer [135], as depicted in Fig. 6.2. As a layer in a neural network, a single MoE may contain up to hundreds of thousands of smaller networks, referred to as experts. Each expert typically contains a few million parameters. The output from the previous layer sparsely activates some of the experts depending on the output of a gating network. An MoE layer has two distinguishing features. Firstly, an MoE layer has a large number of branches, i.e. experts running in parallel, resulting in a non-linear graph topology. Secondly, a single MoE layer may contain up to hundreds of billions of parameters. While a breath-first scheduling strategy as used in TensorFlow maximizes parallelism by executing as many branches in parallel as possible, it may require a massive amount of memory. Therefore, we desire a graph partitioning strategy that applies to non-linear graphs consisting of the finer-grained operations within each individual layer.

Fig. 6.4a plots the memory consumption during a single mini-batch of training a Transformer w/ MoE model, which uses MoE as its feed-forward layer instead of the original dense-relu-dense layer. In this example, this model contains 12 MoE layers in total, with 32 experts per MoE layer and 2 million parameters per expert, resulting in a model with over 800 million parameters. TensorFlow reaches its peak memory at the end of the forward propagation, at around 400ms.

6.3.2 Partitioned Execution and Memory Swapping

Finding a graph execution schedule that minimizes memory consumption is an NP-complete problem [107]. One heuristic to constrain the memory consumption of a non-linear graph is to topologically sort the graph nodes and execute them sequentially in the topologically sorted order. However, sequentially executing all nodes restricts parallelism. We desire a solution that executes some operations in parallel to exploit available compute power while achieving bounded memory usage. Instead, we propose to run partitions of the graph rather than single nodes sequentially, which exploits parallelism within a partition with a bounded number of nodes. For example, as shown in Fig. 6.3, we may partition the MoE layer into 2
Partitioned Execution. Based on these insights, we present a best-effort algorithm (Alg. 4) that partitions the computation graph of each device. The partitioned computation graph executes at most two partitions at a time while maximizing parallelism within each individual partition. Alg. 4 is applied to the optimized execution graph in TensorFlow before Send andRecv operations are added. Alg. 4 essentially performs a depth-first traversal of each device’s computation graph and assigns nodes to fixed-size partitions according to the traversal order. Depth-first traversal makes best effort to consume intermediate results as soon as they are produced, instead of holding them in memory for longer durations.

A graph partition may generate intermediate results that are consumed by partitions that are many sequential steps away in the computation schedule. TensorFlowMem temporarily offloads the intermediate result tensors to host memory and prefetches them by adding SwapOut and SwapIn nodes. Because TensorFlowMem executes graph partitions in a pre-determined order, intermediate results that are not needed by the next \( K \) (by default, we use \( K = 2 \) in the following discussion and our experiments) partitions can be swapped out to host memory with low probability of interfering with computation. Similarly, TensorFlowMem prefetches a tensor from host memory when it’s needed by partitions that will be executed next. TensorFlowMem adds SwapOut and SwapIn nodes to proper partitions based on the execution order so offloading and prefetching are performed at the right time.

Executor. TensorFlowMem revises TensorFlow’s operation scheduler to ensure at most 2 partitions are executed at the same time. TensorFlowMem keeps track of the number of operations in each graph partition and the number of completed operations in each partition. The TensorFlowMem executor does not schedule any operation from partition \( t + 2 \) until all operations from partition \( t \) have been completed. The ready-to-execute operations from partition \( t + 2 \) are buffered and scheduled as soon as partition \( t \) completes.

Fig. 6.4b depicts the memory consumption of TensorFlow with partitioned graph execution and memory swapping using a partition size of 20 operations. We observe the peak
Figure 6.4: Understanding TensorFlow Memory Consumption: Transformer w/ MoE
Algorithm 4: Partition the computation graph

```
input : graph, devices, partSize
output: graph
perDevReadyStack ← EmptyMap();
perDevPartNum ← EmptyMap();
perDevPartSize ← EmptyMap();
while Not all graph nodes are assigned to some partition do
    forall device ∈ devices do
        if perDevReadyStack[device] is not empty then
            node ← perDevReadyStack[device].pop();
            node.partition = perDevPartNum[node.device];
            perDevicePartSize[node.device] += 1;
            if perDevPartSize[node.device] == partSize then
                perDevPartSize[node.device] = 0;
                perDevPartNum[node.device] += 1;
            forall suc ∈ node.successors do
                suc.numReadyInputs += 1;
                if suc.numReadyInputs == suc.inputSize then
                    perDevReadyStack[suc.device].push(suc);
```

memory consumption is reduced to 6.8GB from 9.5GB. The reduced memory consumption
does come with a runtime overhead of $2.5 \times$ in terms of time per mini-batch, due to the re-
duced parallelism and additional data communication between GPU and CPU. Intuitively
reducing the partition size reduces memory consumption and increasing the partition size
increases parallelism. Tuning the partition size gives a different trade-off between memory
consumption and computation throughput.

6.3.3 Operation Placement

A computation graph may contain Variable and Constant nodes that are stateful opera-
tions. Those operations contain an internal buffer that is allocated as a persistent tensor
and is not deallocated until graph destruction. Typically, Variables and Constants are de-
defined by the application program. These nodes are typically packaged with computation
operations that use them as an integral building block by higher level programming inter-
faces, such as Keras [40]. When the application program places the computation operation
on GPUs, Variables and Constants are implicitly placed on GPUs as well due to limited
programming flexibility. Constant folding folds a subgraph into a Constant operation. The
generated Constant operation are placed on the same computing device as the computation
operations.

Variable and Constant nodes may consume a considerable amount of memory espe-
cially when the neural network contains a large number of parameters. Based on this ob-
TensorFlowMem places Variable and Constant nodes on CPU and loads their value to GPU when needed.

(a) TensorFlow inserts one pair of Send and Recv operations for all operations that use the same Variable W1.

(b) TensorFlowMem inserts one pair of Send and Recv for each operation that uses Variable W1.

Figure 6.5: Placement optimization.

**Separate communication.** As shown in Fig. 6.5a, TensorFlow inserts one pair of Send and Recv operations for all operations that use the same Variable or Constant placed on a different device. The received tensor value is used by multiple operations, for example, in the forward pass as well as the backward pass of the same operation. Moreover, since Send depends only on the Variable to be sent, the Variables’ value are received at the beginning of a mini-batch computation, regardless of when they are used, and buffered until the last usage is completed.

To reduce memory consumption, TensorFlowMem inserts distinct pairs of Send and Recv operations for different operations that use the same Variable or Constant. TensorFlowMem assigns the inserted Send and Recv operations to the partition that is shortly before the partition that needs the Variable or Constant, so they are executed shortly before the data value is needed without blocking the computation. As shown in Fig. 6.4c, the placement optimization further reduces peak memory consumption to 3.3GB. The effectiveness of our techniques is also observed for neural networks that do not contain a MoE layer such as Transformer [149]. However, the benefit from operation placement is relatively small for models with small number of parameters.
6.3.4 Alternative Graph Partitioning Strategies

An intuitive idea to recover the layer-wise structure of a fine-grained computation graph is to group operations into partitions based on their height. For long, thin graphs that are similar to a linear chain, such partitioning strategy may be effective. However, for graphs with many branches, this strategy may result in abundant intermediate results across partitions and incur high communication overhead. Here we present two graph partitioning strategies based on node heights and experimentally evaluate their performance.

All TensorFlow computation graphs contain a dummy SOURCE node from which the computation starts and a dummy SINK node at which the computation ends. For each node in the computation graph, we define its depth to be the length of the longest path from this node to the SINK node. We do not simply assign nodes that have the same depth to the same partition since it results in partitions that have too many operations. We propose two depth-based graph partitioning strategies below. Similar to Alg. 4, they traverse the computation graph and assign visited nodes to partitions up to a partition size threshold. During traversal, Alg. 4 prioritizes visiting the last node that becomes ready to run, and the depth-based strategies prioritize nodes based on their depth.

**Depth-guided traversal.** In this strategy, we traverse the graph in the order of decreasing node depth. In essence, this approach creates graph partitions by grouping nodes that have close enough depth while ensuring partitions have balanced and bounded number of operations.

**DFS w/ depth-based prioritization.** In this strategy, we traverse the graph in depth-first order but prioritize visiting nodes with a higher depth.

![Graph Partition Strategies](Figure 6.6: Comparing graph partitioning strategies: DFS vs. Depth (depth-guided traversal) vs. DFS-Depth (DFS w/ depth-based prioritization)).

We empirically compare the three graph partitioning strategies using a Transformer w/ MoE model implemented on Mesh-TensorFlow. The model contains 64 experts per MoE layer and runs using a batch size of 16. Vanilla TensorFlow runs out of memory in this setting. We observe that the depth-guided traversal results in the worst runtime and peak
6.3.5 The Effect of Graph Partition Size

We run a Transformer model implemented on Mesh-TensorFlow to demonstrate the effect of graph partition size. As shown in Fig. 6.7, a larger partition size consumes more memory since there are more operations in each partition. A larger partition also leads to a high execution speed because there are fewer partitions and few inter-partition data movements. For any given problem, the optimal partition size depends on the available memory on the GPU device.

6.4 Evaluation

TensorFlowMem is implemented in TensorFlow v1.12.0 which we use as the baseline. In this section, we present a comprehensive experimental evaluation of TensorFlowMem. All the experiments are conducted in a private cluster, where each machine has a 16-core Intel Xeon E5-2698B v3 processor with hyper-threading, 64GB of DRAM and a Titan X GPU with 12GB device memory. The machines are interconnected with 40Gbps Ethernet.

---

**Figure 6.7:** The effect of graph partition size

memory consumption because the large amount of intra- and inter-partition intermediate results. This result suggests that the computation graph is likely to contain a large number of branches. DFS achieves better performance than DFS w/ depth-based prioritization, suggesting that prioritizing visiting the last node that becomes ready to run is more likely to consume the intermediate results.
### Table 6.3: Details of the benchmark implementations

<table>
<thead>
<tr>
<th>Model</th>
<th>Application</th>
<th>Dataset</th>
<th>Source Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformer</td>
<td>Machine Translation</td>
<td>Wmt32K</td>
<td>Tensor2Tensor [65]</td>
</tr>
<tr>
<td>TransformerMoE</td>
<td>Machine Translation</td>
<td>Wmt32K</td>
<td>Tensor2Tensor [65]</td>
</tr>
<tr>
<td>ResNet</td>
<td>Image Classification</td>
<td>ImageNet1K</td>
<td>Official TensorFlow Models [66]</td>
</tr>
<tr>
<td>WGAN-GP</td>
<td>Image Generation</td>
<td>ImageNet-small</td>
<td>TBD suite [143]</td>
</tr>
<tr>
<td>Mozilla DeepSpeech</td>
<td>Speech Recognition</td>
<td>Common Voice v2.0</td>
<td>Mozilla [119]</td>
</tr>
</tbody>
</table>

### 6.4.1 Methodology and Summary of Results

**Evaluation Objectives.** Our goal is to quantify the effectiveness and benefit of our techniques. First of all, we quantify TensorFlowMem's memory savings and runtime overhead. The key benefit of TensorFlowMem is that it enables ML training applications that would otherwise be impossible on TensorFlow without requiring additional hardware. We quantify this benefit by scaling the mini-batch size, model size and sequence length in an RNN.

**Benchmarks.** Unlike previous related work [35, 153] which are evaluated only on a narrow set of benchmarks such as CNNs and RNNs, we evaluate TensorFlowMem on a wide range of popular and important deep neural networks with various features, including convolution, recurrence, attention, MoE and GANs, as shown in Table 6.2. TensorFlowMem requires no modifications to the application program. Details of our benchmark applications can be found in Table 6.3. The original Transformer [149] uses Dense-Relu-Dense (DRD) as its feedforward layer. Transformer w/ MoE replaces DRD with MoE.

Mozilla DeepSpeech as an RNN is implemented by statically unrolling the sequence. A training dataset may contain sequences of variable length and thus memory consumption and per-mini-batch execution time vary from mini-batch to mini-batch. For the purpose of comparison, we fixed the sequence length, chopping and padding if necessary during execution. TensorFlow acquires the whole GPU memory and manages memory allocation internally. We implemented a memory profiler in TensorFlow to measure the memory that is actually occupied by graph computation.

We summarize our key evaluation results below:

1. Across 5 neural networks, TensorFlowMem reduces peak memory consumption by 64.8% on average (up to 87.9%) with an average 2.2× overhead in run time.

2. For models that do not have excessive numbers of parameters, e.g., ResNet, Transformer, DeepSpeech and GAN, partitioned graph execution and memory swapping together reduce peak memory consumption by 60% in average with a 1.55× runtime overhead.

3. TensorFlowMem enables training a ResNet model with 1916 layers (limited by host memory) in contrast to 504 layers on TensorFlow using a single GPU.

4. TensorFlowMem, scales Transformer w/ MoE to 2.5 Billion parameters using a single GPU in contrast to 0.7 Billion parameters on TensorFlow.
Figure 6.8: Ablation study on a single GPU. Vanilla represents vanilla TensorFlow; +Partition represents TensorFlow with partitioned execution and memory swapping; +Placement represents placement optimization on top of +Partition.

5. TensorFlowMem trains statically unrolled RNNs on 2× longer sequences with a 1.35× runtime overhead.

6. TensorFlowMem can train a Transformer w/ MoE model with 8.1 Billion parameters on 4 machines, while TensorFlow fails to scale beyond 2.7 Billion parameters.

6.4.2 Effectiveness of Individual Techniques

Methodology. We evaluate the effectiveness of individual techniques, i.e., partitioned graph execution w/ memory swapping (i.e., Sec. 6.3.2) and placement optimization (i.e., Sec. 6.3.3), by comparing TensorFlowMem with TensorFlow in terms of peak memory consumption.
and time per mini-batch. We conduct this ablation study on Transformer (with both DRD and MoE versions), ResNet-152, WGAN-GP and DeepSpeech on a single Titan X GPU. For DeepSpeech, the sequence length is fixed to 165. All models are trained with a partition size of 20 except for DeepSpeech, which uses a partition size of 5. A smaller partition size (5 vs. 20) allows TensorFlowMem to more aggressively reduce memory consumption with a relatively small overhead for DeepSpeech. All models are executed using the largest possible mini-batch that TensorFlow can support on a 12GB GPU.

Fig. 6.8a shows the peak memory consumption and Fig. 6.8b shows the time per mini-batch normalized to the baseline (i.e., vanilla TensorFlow). We compute the average peak memory and average per-mini-batch time across all models, i.e., Avg, and across all models except for Transformer w/ MoE, i.e., Avg-NoMoE. We separate Transformer w/ MoE from other models because MoE contains a distinctly large number of parameters compared to other models.

<table>
<thead>
<tr>
<th></th>
<th>Vanilla</th>
<th>+Partition</th>
<th>+Placement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Consumption</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>10.8</td>
<td>4.8</td>
<td>3.8</td>
</tr>
<tr>
<td>Average w/o MoE</td>
<td>11.0</td>
<td>4.4</td>
<td>4.0</td>
</tr>
<tr>
<td>Runtime Overhead</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>1</td>
<td>1.8</td>
<td>2.2</td>
</tr>
<tr>
<td>Average w/o MoE</td>
<td>1</td>
<td>1.55</td>
<td>1.85</td>
</tr>
</tbody>
</table>

Table 6.4: Average memory consumption and runtime overhead across all models.

**GPU Memory reduction.** From Fig. 6.8a, we observe that TensorFlowMem reduces GPU memory consumption by 65% on average across all models and by 87% for WGAN-GP. While the placement optimization further reduces memory consumption by 36.8% on top of partitioned execution and memory swapping for Transformer w/ MoE, it only brings a small benefit to the other models that have relatively small number of parameters.

**Runtime overhead.** Fig. 6.8b shows that TensorFlowMem incurs a $2.2 \times$ slowdown in terms of time per mini-batch in average. The runtime overhead vary greatly across models. It can be as large as $3.4 \times$ for Transformer w/ MoE and as small as 30% for DeepSpeech. Note that for models without a huge number of parameters, we achieve a 65% memory reduction with a 55% runtime overhead using only partitioned graph execution and memory swapping.

<table>
<thead>
<tr>
<th>Model</th>
<th>TensorFlow</th>
<th>TensorFlowMem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformer</td>
<td>11264 (words)</td>
<td>25600 (words)</td>
</tr>
<tr>
<td>Transformer w/ MoE</td>
<td>4 (sentence pairs)</td>
<td>40 (sentence pairs)</td>
</tr>
<tr>
<td>ResNet-152</td>
<td>57 (images)</td>
<td>186 (images)</td>
</tr>
<tr>
<td>WGAN-GP</td>
<td>90 (images)</td>
<td>324 (images)</td>
</tr>
</tbody>
</table>

Table 6.5: The maximum supported mini-batch size by both systems
Table 6.6: Throughput using the maximum supported mini-batch size.

<table>
<thead>
<tr>
<th>Model</th>
<th>TensorFlow</th>
<th>TensorFlowMem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformer</td>
<td>8731 (words / sec)</td>
<td>3699 (words / sec)</td>
</tr>
<tr>
<td>Transformer w/ MoE</td>
<td>4.6 (s. pairs /sec)</td>
<td>5.2 (s. pairs / sec)</td>
</tr>
<tr>
<td>ResNet-152</td>
<td>50 (images / sec)</td>
<td>27 (images / sec)</td>
</tr>
<tr>
<td>WGAN-GP</td>
<td>7.9 (images / sec)</td>
<td>4.5 (images / sec)</td>
</tr>
</tbody>
</table>

6.4.3 Training w/ Larger Mini-Batches

By reducing GPU memory consumption, TensorFlowMem allows applications to train models using larger batch sizes. While the common wisdom for using larger batch size is to improve computation throughput. There are cases where larger mini-batch sizes lead to higher model accuracy, for example, as shown in GPipe [80]. TensorFlowMem provides users an additional degree of freedom for using larger mini-batch sizes. Table 6.5 shows the largest mini-batch supported by TensorFlow and TensorFlowMem and Table. 6.6 shows the computation throughput using the maximum batch size. For Transformer, ResNet and WGAN-GP, we observe that TensorFlowMem enables training 2.2 – 3.6× larger mini-batch size with a 43% – 58% slowdown in computation throughput. TensorFlowMem supports a 10× larger mini-batch size for Transformer w/ MoE, which contains an order of magnitude more parameters than other models. TensorFlowMem also improves the computation throughput for Transformer w/ MoE by 6%. TensorFlowMem enables a larger range of possible mini-batch sizes compared to TensorFlow, allowing applications to tune mini-batch size for high model quality, especially for models with a large number of parameters.

6.4.4 Training Larger Models

It has been shown that larger models achieve better accuracy across different application domains [72, 73, 135]. The key benefit of TensorFlowMem is that it enables training larger models that is otherwise impossible without using additional hardware. In this section, we demonstrate that TensorFlowMem enables training ResNet with 3.8× more layers and Transformer w/ MoE with 4.4× more parameters.

**Deeper ResNet.** Similar to previous work [35, 153], we increase the model capacity of ResNet by scaling its number of layers. Specifically, we follow SuperNeuron [153] and increase the number of the third block. Our result is present in Table 6.7. Using the same mini-batch size of 16 images, TensorFlowMem scales to 1916 layers while TensorFlow scales to only 504 layers. Using a mini-batch size of 32, TensorFlowMem scales to 1001 layers. Moreover, TensorFlowMem fails to scale to deeper ResNet because of running out of host memory.

**Mixture of Experts.** We evaluate TensorFlowMem using Transformer w/ MoE. Table 6.8 shows the maximum number of experts per MoE layer which can trained using TensorFlow and TensorFlowMem. TensorFlowMem is able to train 48 experts for MoE layer, which is
4× as many as vanilla TensorFlow.

<table>
<thead>
<tr>
<th>System</th>
<th>Mini-batch size</th>
<th>#Layers</th>
<th>#Param.</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>TensorFlow</td>
<td>16</td>
<td>504</td>
<td>172 Million</td>
<td>11.43</td>
</tr>
<tr>
<td>TensorFlowMem</td>
<td>16</td>
<td>1916</td>
<td>697 Million</td>
<td>1.76</td>
</tr>
<tr>
<td>TensorFlowMem</td>
<td>32</td>
<td>1001</td>
<td>385 Million</td>
<td>4.04</td>
</tr>
</tbody>
</table>

Table 6.7: Maximum ResNet model size that can be trained on a single Titan X GPU and computation throughput with different mini-batch size.

<table>
<thead>
<tr>
<th>System</th>
<th>#Experts</th>
<th>#Param.</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>TensorFlow</td>
<td>12 / MoE</td>
<td>0.66 Billion</td>
<td>7.8 pairs / sec</td>
</tr>
<tr>
<td>TensorFlowMem</td>
<td>48 / MoE</td>
<td>2.5 Billion</td>
<td>1.2 pairs / sec</td>
</tr>
</tbody>
</table>

Table 6.8: Maximum number of experts that can be trained on a single TitanX GPU. We use a batch size of 8 and graph partition size of 200.

<table>
<thead>
<tr>
<th>Sequence Length</th>
<th>100</th>
<th>200</th>
<th>400</th>
<th>500</th>
<th>800</th>
</tr>
</thead>
<tbody>
<tr>
<td>TensorFlow</td>
<td>1.15</td>
<td>2.3</td>
<td>4.64</td>
<td>OOM</td>
<td>OOM</td>
</tr>
<tr>
<td>TensorFlowMem</td>
<td>1.56</td>
<td>3.03</td>
<td>6.03</td>
<td>-</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 6.9: RNN training: time per mini-batch (seconds) for different input sequence length.

### 6.4.5 Longer Recurrence Sequences

For RNNs, the sequence length is often limited by GPU memory size. TensorFlowMem enables training RNNs on longer sequences with a small runtime overhead, which we demonstrate using Mozilla DeepSpeech, which is a statically unrolled RNN. Our experiments use a mini-batch size of 128 sentences and the partition size of TensorFlowMem is set to 5. Table 6.9 shows that TensorFlowMem can train DeepSpeech on sequences of length 800 while TensorFlow fails beyond sequence length of 400. Similar to ResNet, TensorFlowMem fails to scale to longer sequences due to the limited host memory. On the same sequence length, TensorFlowMem shows a runtime overhead of roughly 35%.

### 6.4.6 Distributed Model-Parallel Training

Next, we show that TensorFlowMem’s memory optimization techniques can be directly applied to the distributed model-parallel setting as well. By utilizing 4 GPU machines, even larger models with more experts can be trained. Table 6.10 highlights these results. For all experiments, we split only the experts dimension of tensors across the 4 nodes, and replicate all other dimensions. In the distributed setting, the memory optimizations in TensorFlowMem are simply applied independently on each node. TensorFlowMem is able to train a model with 8.1 billion parameters, 3× as many as vanilla TensorFlow in the same distributed setting.

2We ran TensorFlow both with and without the Grappler memory swapping pass, but obtained the same result both times.
Table 6.10: Maximum number of experts that can be trained on 4 nodes each with a single TitanX GPU. We use a batch size of 8 and graph partition size of 200.

### 6.4.7 Comparison with Related Work

In this section, we compare TensorFlowMem with the TensorFlow’s native Grappler Memory Optimizer and SuperNeurons [153] on scaling model capacity.

**Grappler Memory Optimizer.** The Grappler Memory Optimizer statically estimates the graph execution time and memory consumption and invokes a swapping pass when and only when the estimated memory consumption of at least one GPU exceeds its memory capacity. The static estimation results also provides the start and end times of each operation and the memory footprint of each tensor. The swapping pass relies on this information to swap tensors in and out around the memory peak to avoid running out of memory. According to operations’ start and end time, control dependencies are added to the swap operations so that they are executed at appropriate times to avoid stalling computation.

<table>
<thead>
<tr>
<th>Model</th>
<th>Mini-batch Size</th>
<th>Grappler Memory Optimizer</th>
<th>TensorFlowMem</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet</td>
<td>16 images</td>
<td>504 layers</td>
<td>1916 layers</td>
</tr>
<tr>
<td>Transformer w/ MoE</td>
<td>8 sentence pairs</td>
<td>12 experts / MoE</td>
<td>48 experts / MoE</td>
</tr>
</tbody>
</table>

Table 6.11: Largest model configuration supported by Grappler Memory Optimizer and TensorFlowMem.

<table>
<thead>
<tr>
<th>Model</th>
<th>Model Configuration</th>
<th>Predicated OOM</th>
<th>Actual OOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet</td>
<td>1001 layers</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>Transformer w/ MoE</td>
<td>48 experts / MoE</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Transformer</td>
<td>N/A</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

Table 6.12: Grappler memory optimizer: simulator prediction and effectiveness.

Table 6.11 shows the largest ResNet and MoE model that TensorFlow (with Grappler Memory Optimizer enabled) can run. Surprisingly, we found the Grappler Memory Optimizer does not improve upon TensorFlow in terms of model capacity. In order to understand why Grappler Memory Optimizer fails we analyzed three large models that result in OOM when executing on TensorFlow with Grappler Memory Optimizer (Fig. 6.12). Static estimation fails to predict the OOM error for the ResNet mode with 504 layers. While static estimation correctly predicts the OOM error and the Grappler Memory Optimizer indeed adds swap operations for the two Transformer-based models, training still runs out of memory on GPU. Note that all three models can execute with TensorFlowMem without running out of memory. We speculate that the Grappler Memory Optimizer’s failure is due to the discrepancy between static estimation and actual execution. Also, when the
OOM is due to a large fanout, it cannot be mitigated without intentionally delaying some computation.

**SuperNeurons.** SuperNeurons also relies on memory swapping to scale model capacity with limited GPU memory. It effectively scales the mini-batch size for several convolutional neural networks and enables training much deeper ResNet [153]. However, it is a prototype implementation that does not support TensorFlow API. We compare our results with SuperNeurons’ as report by Wang et al. [153]. Using a single GPU with 12 GB of memory, SuperNeurons scales ResNet to 1920 layers while TensorFlowMem scales ResNet to 1916 layers (on a different GPU that has the same memory size). When training on ImageNet, SuperNeurons scales ResNet-152 to a maximum batch size of 176 images while TensorFlowMem scales the maximum batch size to 186 images. Note that SuperNeurons also incurs none trivial overhead at when the model’s memory footprint exceeds the GPU memory capacity. For examples, the computation throughput drops by nearly 30% when scaling ResNet-152 to a mini-batch size of 80 images. TensorFlowMem suffers a 46% overhead when scaling mini-batch size to 186 images. SuperNeurons was not evaluated on other DNN models besides CNNs.

### 6.5 Memory-Efficient Application Implementation on TensorFlow

Implementation of the DNN model affects its memory consumption. In this section, we present two application implementation guidelines that can enable scaling to larger models, especially when using TensorFlowMem. We demonstrate the effectiveness of these guidelines by modifying Mesh-TensorFlow which is a library on top of TensorFlow for model-parallel training and the MoE implementation that’s based on Mesh-TensorFlow. Together with TensorFlowMem, these changes enable training a $7.5 \times$ larger MoE model than the original implementations using TensorFlow.

#### 6.5.1 Application Implementation Guidelines

**Partition a large operation into smaller operations.** TensorFlowMem’s partitioned execution and memory swapping take each operation as an atomic unit. They fail to overcome the device memory constraint when a single operation consumes or produces a tensor that occupies too large an amount of memory. The application program can alleviate this problem by partitioning a big operation that produces a few big tensors into many small operations that produce many small tensors, permitting graph partitioning and memory swapping to work at a finer granularity. There exist many opportunities to parallelize larger operations. Tensors that are too small may lead to inefficient use of massively parallel processing units, e.g., GPUs. Therefore, it is critical to partition tensors to the proper size. While we rely on users to manually, we expect future work may improve upon manual partitioning using automated search.

**Allocate tensors proportionally to input size.** In some DNNs, the size of the input to some operations is dynamically determined. While it is convenient and fast to allocate a tensor...
of fixed size to hold the largest possible input, this approach can waste a large amount of memory. This is particularly wasteful when a large tensor is dynamically split into smaller tensors of variable size, such as in MoE. In this case, the application either allocates the maximum possible size for each dynamically allocated tensor, which is the input size, or drops some of the input values. In case of OOM, it is better for applications to suffer a small throughput penalty and allocate memory proportionally to input size, for example, by using the sparse gather operation.

6.5.2 Over-Partitioning Operations in Mesh-TensorFlow

Mesh-TensorFlow [136] is a framework for model-parallel training on top of TensorFlow. It partitions the large operations of a computation graph and place the them on different computing devices. However, the number of partitions a Mesh-TensorFlow tensor may have is upper-bounded by the number of physical computing devices. Thus it may still result in a large operation that consumes a large fraction of GPU memory by itself. To address this issue, we designed and implemented Virtual Mesh-TensorFlow (VMesh-TensorFlow), an extension to Mesh-TensorFlow, which partitions a large operation bound to a specific device into smaller operations. VMesh-TensorFlow works in conjunction with partitioned graph execution and memory swapping to enable execution of models whose memory consumption exceeds the overall memory capacity of all of the devices in the mesh.

Background: Mesh-TensorFlow

Like TensorFlow, Mesh-TensorFlow requires applications to define a computation graph. Tensors in a Mesh-TensorFlow computation graph have named dimensions and different tensors may share the same name along some of their dimensions. Mesh-TensorFlow additionally requires the application program to define a mesh shape and a mesh layout. A mesh shape defines a multi-dimensional grid composed of computing devices (e.g., GPUs), and a mesh layout is a mapping from named tensor dimensions to mesh dimensions. A Mesh-TensorFlow is lowered to a TensorFlow graph by partitioning the Mesh-TensorFlow tensors along dimensions that are mapped to a mesh dimension and replicating the tensors along other mesh dimensions. In this way, a large tensor may be split across multiple computing devices as well as the operations that generate and produce the tensor.

Virtual Mesh-TensorFlow

Mesh Over-partitioning. In Mesh-TensorFlow, the number of nodes in the mesh layout equals the number of physical devices. For example, a mesh with shape \((3, 2)\) corresponds to 6 physical devices logically arranged in a \(3 \times 2\) grid. VMesh-TensorFlow additionally introduces a device shape that specifies a virtual grid for each device (we refer to the mesh shape in Mesh-TensorFlow as cluster shape for clarity). VMesh-TensorFlow requires the device shape and the cluster shape to have the same number of dimensions. Besides partitioning tensors according to the cluster shape, VMesh-TensorFlow further partitions the tensors allocated for each device according to the device shape using the same logic. In other words, VMesh-
TensorFlow partitions tensors according to a mesh shape that is the element-wise product of the cluster shape and device shape (Fig. 6.9). Over-partitioning the mesh enables large tensors and operations to be split into finer-grained partitions on each physical device, and swapped into device memory individually. In our implementation, the Mesh-TensorFlow graph is simply compiled using the virtual mesh instead of the original physical mesh, with no changes required to the application code. Similar techniques for using over-partitioning to abstract away physical devices have been explored in prior work [17, 126].

**Graph De-duplication.** Although mesh over-partitioning enables finer-grained splitting of tensors and operations, it by itself introduces significant memory and performance overhead. In Mesh-TensorFlow, if a tensor is not split across a mesh dimension, then it is replicated across it. However, if that mesh dimension is over-partitioned by a factor of $N$ (ie. its device shape dimension has size $N$), then the tensor (and all operations using the tensor) will be replicated $N$ times on each physical device.

To overcome this limitation, we implement a de-duplication pass which runs when the VMesh-TensorFlow graph is compiled to a TF graph. The de-duplication pass finds any tensors, variables, and operations which are replicated multiple times on a single device, and replaces their uses with references to a single “master” copy. The other unused copies are then pruned out by TensorFlow before execution.

**6.5.3 Memory Efficient MoE Implementation**

The techniques presented thus far in this section are generic framework improvements which can apply to existing model code. On the other hand, the model creator may achieve

---

1We note that the duplication issue is specific to Mesh-TensorFlow, other model-parallel frameworks may not have the same limitation.
further memory efficiency by writing their model in a way which is aware of TensorFlowMem and VMesh-TensorFlow. As mentioned in Sec 6.5.2, atomic operations on large tensors are a limiting factor for partitioned graph execution and memory swapping. Although VMesh-TensorFlow can alleviate this issue by over-partitioning large tensors into finer-grained slices, the model creator can often better control these limiting factors by choosing different operations or constructing different computation graphs. How this can be done varies from model to model, and we will present one specific optimization we made to the MoE implementation on Mesh-TensorFlow.

**Increasing Sparsity in Mesh-TensorFlow MoE.** Although the experts in MoE layers are sparsely activated, the publicly-available implementation using Mesh-TensorFlow uses a few large dense operations [65]. In this implementation, all inputs for all experts are copied into a single large tensor, and two `einsum` operations are used to compute the expert outputs for all inputs at once. However, in order to account for different experts having different numbers of inputs, a large fraction of the combined input tensor (and by extension intermediate tensors during the experts computation) may be padded with zeros, consuming unnecessary amounts of memory. Instead, we changed the MoE implementation to collect the inputs needed for each expert using a sparse `gather` operation, so that the need for padding a large dense tensor is eliminated. Doing this effectively exploits the sparsity of the experts activated by each input sample, sending only the necessary data through the experts computation.

### 6.5.4 Evaluation

**Mixture of Experts.** We evaluate VMesh-TensorFlow using Transformer w/MoE. Table 6.13 presents the maximum number of experts per MoE layer that can be trained using VMesh-TensorFlow on top of TensorFlowMem. As presented in Table 6.8, the original Mesh-TensorFlow running on TensorFlowMem is able to train 48 experts for MoE layer. We use VMesh-TensorFlow to partition the experts into 4 groups per MoE layer, which enables TensorFlowMem to swap out the large tensors in each MoE layer at a finer granularity. Overall, VMesh-TensorFlow on TensorFlowMem is able to train a model with 56 experts per MoE layer (totaling 2.9 billion parameters) on a single GPU, 4.4× larger than vanilla TensorFlow. We also find that the throughput decreases roughly linearly with respect to the number of experts when scaling up the MoE layers using TensorFlowMem and VMesh-TensorFlow.

<table>
<thead>
<tr>
<th>System</th>
<th>#Experts</th>
<th>#Param.</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>TensorFlow</td>
<td>12 / MoE</td>
<td>0.66 Billion</td>
<td>7.8 pairs / sec</td>
</tr>
<tr>
<td>TensorFlowMem</td>
<td>48 / MoE</td>
<td>2.5 Billion</td>
<td>1.2 pairs / sec</td>
</tr>
<tr>
<td>+VMesh-TensorFlow</td>
<td>56 / MoE</td>
<td><strong>2.9 Billion</strong></td>
<td>0.87 pairs / sec</td>
</tr>
</tbody>
</table>

Table 6.13: Maximum number of experts that can be trained on a single TitanX GPU. We use a batch size of 8 and graph partition size of 200. For VMesh-TensorFlow, we split the batch and experts dimensions of all tensors across a virtual mesh of size 4.
### Distributed Model-Parallel Training

<table>
<thead>
<tr>
<th>System</th>
<th>#Experts</th>
<th>Param. Size</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>TensorFlow</td>
<td>52 / MoE</td>
<td>2.7 Billion</td>
<td>8.5 pairs /sec</td>
</tr>
<tr>
<td>TensorFlowMem</td>
<td>160 / MoE</td>
<td>8.1 Billion</td>
<td>0.93 pairs /sec</td>
</tr>
<tr>
<td>+VMesh-TensorFlow</td>
<td>200 / MoE</td>
<td><strong>10.1 Billion</strong></td>
<td>0.52 pairs /sec</td>
</tr>
<tr>
<td>+SparseMoE</td>
<td>400 / MoE</td>
<td><strong>20.2 Billion</strong></td>
<td>1.6 pairs /sec</td>
</tr>
</tbody>
</table>

**Table 6.14**: Maximum number of experts that can be trained on 4 nodes each with a single TitanX GPU. We use a batch size of 8 and graph partition size of 200. For VMesh-TensorFlow and Sparse-MoE, we split the experts dimension of all tensors across a virtual mesh of size 20 (cluster shape of 4 and device shape of 5).

Application-level optimizations in VMesh-TensorFlow and SparseMoE enable training larger models on TensorFlowMem using distributed model parallelism. We demonstrate it using the same settings as used in Sec. 6.4.6 (totally 4 GPU machines), and the results are presented in Table 6.10. As before, we split only the experts dimension of tensors across the 4 machines and replicate all other dimensions.

**TensorFlowMem+VMesh-TensorFlow.** With VMesh-TensorFlow, we over-partition the mesh by a factor of 5, resulting in a virtual mesh of size 20. This means that any tensors with an experts dimension are split into 20 slices, 5 on each node. Combining TensorFlowMem with VMesh-TensorFlow enables the number of experts per MoE layer to be increased to 200, forming a model with over 10 billion parameters.

**TensorFlowMem+VMesh-TensorFlow+SparseMoE.** Lastly, we evaluate the sparse MoE implementation described in Sec 6.5.3. By combining TensorFlowMem, VMesh-TensorFlow, and SparseMoE, we are able to train 400 experts per layer, in a model which has over 20 billion parameters. We also note that the throughput of training increased even when compared with training smaller models without SparseMoE. This is likely due to eliminating redundant computations on padding values in large dense tensors. Similar to ResNet and Transformer w/ MoE on a single machine, we also observe that with larger MoE layers, the host memory rather than GPU memory is exceeded, indicating that even larger models can be trained with an expansion to host memory.

### 6.6 Summary

In this chapter, we present TensorFlowMem that reduce TensorFlow’s GPU memory consumption to enable training substantially larger models in both single GPU and distributed settings. The distributed model-parallel application can be improved to better leverage TensorFlowMem to train even larger models on the same hardware. TensorFlowMem demonstrates that scheduling where and when memory is allocated allows scaling model size with a small overhead. We expect that TensorFlowMem memory optimizations can be improved by leveraging more accurate estimations of the operation execution time and memory footprint.
Chapter 7

Conclusion and Future Directions

7.1 Conclusion

In this thesis, we demonstrate that the domain-specific characteristics of machine learning training can be leveraged to schedule hardware resources to improve the training speed and train larger models. Specifically, we present a mechanism for scheduling network communication that improves the training convergence speed by up to $5 \times$, a computation scheduling mechanism that further improves training speed, while consuming lower network bandwidth and substantially reducing programmer effort, and a memory scheduling mechanism that enables training up to $7.5 \times$ larger models on the same hardware with acceptable performance overhead.

The ideas that we explore, such as value-based prioritization, scheduling computation to avoid conflicting accesses, and scheduling memory allocation based on computation, are generalizable to broad machine learning applications. However, their implementations and effectiveness depend on the specific machine learning model, algorithm, as well as hardware architecture. Machine learning is a fast advancing field. New models and improvements to learning algorithms are being invented at a high pace. The success of machine learning motivates new software systems and hardware architectures to be developed to better serve this important workload, which in turn makes it easier to develop and experiment with new machine learning models and algorithms.

With the fast advancements in machine learning techniques, software systems and hardware architectures, the research focus on machine learning systems is shifting from network and storage I/O to computation. Existing works on computation optimization are focused on the memory bandwidth bottleneck, improving cache hit rate, exposing opportunities for parallelization and reducing the overhead of interacting with hardware accelerators.

7.2 Future Directions

In this section we discuss some opportunities and challenges in improving upon the techniques we present, generalizing them to newer machine learning models and across hard-
There exist many techniques to reduce GPU memory consumption and thus enable training larger models and they all involve a trade-off between memory consumption and computation throughput or accuracy. Ultimately, our goal is to maximize training speed subject the device memory constraints. This problem is hard and in this section we discuss our vision in tackling this problem.

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Trade-Off</th>
<th>Notable Prior Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduling</td>
<td>Degree of parallelism</td>
<td>TensorFlowMem</td>
</tr>
<tr>
<td>Gradient checkpointing</td>
<td>Recomputation</td>
<td>[35, 146]</td>
</tr>
<tr>
<td>Anti-constant folding</td>
<td>Recomputation</td>
<td>TensorFlowMem</td>
</tr>
<tr>
<td>Memory swapping</td>
<td>Communication</td>
<td>TensorFlowMem, SuperNeurons [153]</td>
</tr>
<tr>
<td>Device placement</td>
<td>Communication</td>
<td>TensorFlowMem</td>
</tr>
<tr>
<td>Lossless computation</td>
<td>Computation</td>
<td>Gist [81]</td>
</tr>
<tr>
<td>Quantization</td>
<td>Accuracy</td>
<td>[41, 42, 98, 114]</td>
</tr>
</tbody>
</table>

Table 7.1: Summary of memory optimization techniques and their trade-offs[81].

Maximizing training speed subject to device memory constraints is a particularly hard problem for the following reasons:

- **Extremely large search space.** As shown in Table 7.1, there exist many techniques to optimize the memory consumption of DNN training and many techniques involve tuning parameters. For example, TensorFlowMem’s partitioned execution introduces a performance-sensitive parameter, partition size. It may also be important to decide which constant subgraph should be folded depending on the size of the result and the computation complexity, and what tensors to place on which devices based on the size of the operation and the communication bandwidth.

- **Interdependence between optimization techniques.** Since the memory optimization techniques often affect each other, the order in which the optimizations are applied is important. The interdependence between operations makes the search space exponentially hard.

- **Hardware- and workload-specific.** The above mentioned techniques involve trade-off between memory consumption and different hardware resources or computation quality. The best configuration of techniques thus depend on the hardware and the DNN model. For example, in case there is abundant bandwidth between host and GPU memory, aggressively placing large tensors on host memory would be more effective than introducing redundant computation.

- **Memory efficient DAG scheduling is hard.** It is an NP-complete problem to schedule a DAG to execute using minimal memory.

- **Accurate simulation is difficult.** Many the above techniques rely on knowing the
size of the tensors and operations’ execution time. While many of them simulate the graph execution to obtain such information, accurate simulation is difficult as we’ve shown in Sec. 6.4.7.

This problem is essentially a search problem over the space of memory optimization techniques.

### 7.2.2 Dynamic Scheduling for Dynamic Control Flow

In a dataflow system, application programs first construct a dataflow graph that describes the computation, and then request the system to execute a subgraph or the whole graph. Although for many neural networks (e.g., AlexNet [95], Inception-v3 [144], and ResNet [72]), the computation can be described by a static acyclic directed graph (DAG) that applies to all data samples, there are many cases where the graph topology varies based on input or parameter values.

#### Recurrent Neural Networks [55]

model sequences of data (e.g., sentences). A recurrent neural network (RNN) repeatedly applies a cell function, such as long-short-term-memory (LSTM) [76], to each element of the sequence. Since sequences may have variable length, the cell function is executed for different number of times for different sequences. A typical approach for expressing RNNs as a static DAG is to statically unroll the sequence for a finite number of steps, padding shorter sequences with empty values and likely chopping longer ones. An alternative approach is to construct a distinct graph for each input sequence, paying the graph construction overhead for each data sample.

#### Recursive Neural Networks [141]

generalize recurrent neural network to model arbitrary topologies. For example, Tree-LSTM [145] models the syntactic tree of a sentence. Since the topology differs from sentence to sentence, Tree-LSTM constructs a distinct static DAG for each sentence. As shown by Xu et al. [165], per-sample graph construction constitutes a significant overhead (over 60% of runtime in some cases). Xu et al. [165] propose to resolve the graph construction overhead by reusing the graph structure that already exists in the dataset instead of programmatic construction, restricting its applicability.

#### Mixture of Experts (MoE) [135]

is an example of conditional computation in neural networks. A MoE layer consists of a gating network and a large number (up to hundreds of thousands) of expert networks. Each data sample sparsely activates a small number of experts as determined by the gating network based on runtime values. Therefore, for an input mini-batch, the input size of each expert is unknown until the gating network has been executed on the mini-batch.

#### Expressing dynamic computation via dynamic control flow

Yu et al. [170] present two dynamic control flow operations `cond` and `while_loop` in TensorFlow that represents conditional and iterative computation respectively.

Recursive (including recurrent) neural networks can be expressed as a while loop iterating over the nodes in a topologically sorted order. As the loop body is represented as
a subgraph in a static DAG, all dynamic instances of the loop body (i.e., iterations) share the same dependency pattern. Therefore, for recursive neural networks, each iteration is conservatively specified to depend on its previous iteration to ensure correct ordering, resulting in a sequential execution, even though some iterations can potentially be executed in parallel. Jeong et al. [83] take advantage of the additional parallelism by introducing a recursion operation into TensorFlow. With recursion, a node recursively invokes the computation function on other nodes and waits until the recursive calls return to continue its execution. This allows a caller to dynamically specify its distinct dependency on the callees, permitting parallel execution of the functions on independent nodes.

The Need for Dynamic Scheduling of Dynamic Control Flow

Existing dataflow-based deep learning systems employ a static schedule derived prior to graph execution. This schedule determines how operations are placed on (possibly distributed) computing devices and compiles each device’s graph partition to an executable program. As discussed earlier in Sec. 2.6, previous works propose to find an efficient schedule using machine learning when the same static computation graph applies to all data samples. However, the effectiveness of this approach is limited when the computation graph depends on input data and parameter values. In this section, we focus on distributed device placement.

Conditional Computation. TensorFlow’s cond is implemented using Switch which forwards an input tensor to one of two subgraphs. MoE generalizes Switch in two ways: (1) the forwarding decision is made separately for each row in the input tensor and (2) each row is forwarded to $K$ out of $N$ subgraphs. Due to MoE’s large size (up to $\sim 131$ billion parameters), existing implementations (e.g., Tensor2Tensor [150] and Shazeer et al. [135]) statically partition the expert networks to different GPUs. Such static placement faces two problems: (1) the memory for a subgraph (e.g., variables) is statically allocated regardless of whether a subgraph is actually executed; (2) the input sizes among different experts can be highly skewed. These issues lead to heavy over-provisioning of GPU memory while wasting GPUs’ precious computing cycles. As reported by Shazeer et al. [135], a MoE layer consisting of 131072 experts requires 128 Tesla K40 GPUs to fit while achieving a computation throughput of 0.3TFLOPS per GPU (Nvidia’s claimed peak throughput is 4.29TFLOPS/GPU). With dynamic scheduling, the system allocates memory for only subgraphs that are executed and may partition an overwhelmingly large input to an expert along with replicating the expert to multiple GPUs to balance load among GPUs.

Iterative and Recursive Computation. TensorFlow creates a frame for each dynamic instance of the while_loop loop body. Operations of different frames may run in parallel as long as their dependencies are satisfied. However, since each operation is statically placed onto one device, all frames of this operation are bound to this device. This can lead to saturating the computing power of a single device, thus missing the additional parallelism, such as observed by Jeong et al. [83]. Previous work on static device placement observes throughput improvement when placing different iterations of a statically unrolled RNN to
different devices [86, 115, 116]. While static scheduling would be prohibitively expensive when different data samples require different graph topology, dynamic scheduling may dynamically schedule different frames to different devices to take advantage of the additional parallelism. Moreover, as recursion is restricted to trees, deep learning systems need a more general approach for precisely capturing the dependency among loop iterations in order to explore parallelism in arbitrary dependency topologies, such as Graph-LSTM [102].
Appendices
Appendix A

Orion Application Program Examples

A.1 Stochastic Gradient Descent Matrix Factorization

```julia
include("/path/to/orion/src/julia/orion.jl")
Orion.set_lib_path("/path/to/orion/lib/liborion_driver.so")

const master_ip = "10.117.1.17"
const master_port = 10000
const comm_buff_capacity = 1024
const num_executors = 64
const num_servers = 1

Orion.glog_init()
Orion.init(master_ip, master_port, comm_buff_capacity, num_executors, num_servers)

const data_path = "file:///path/to/data.csv"
const K = 1000
const num_iterations = 256
const step_size = Float32(0.01)

Orion.@accumulator err = 0
Orion.@accumulator line_cnt = 0

Orion.@share function parse_line(line::AbstractString)
    global line_cnt
    line_cnt += 1
    tokens = split(line, ',')
    @assert length(tokens) == 3
    key_tuple = (parse(Int64, String(tokens[1])),
                   parse(Int64, String(tokens[2])))
    value = parse(Float32, String(tokens[3]))
    return (key_tuple, value)
end
```
Orion.@share function map_init_param(value::Float32)::Float32
    return value / 10
end

Orion.@dist_array ratings = Orion.text_file(data_path, parse_line)
Orion.materialize(ratings)
dim_x, dim_y = size(ratings)
println((dim_x, dim_y))
line_cnt = Orion.get_aggregated_value(:line_cnt, :) +
println("number of lines read = ", line_cnt)
Orion.@dist_array W = Orion.randn(K, dim_x)
Orion.@dist_array W = Orion.map(W, map_init_param, map_values = true)
Orion.materialize(W)
Orion.@dist_array H = Orion.randn(K, dim_y)
Orion.@dist_array H = Orion.map(H, map_init_param, map_values = true)
Orion.materialize(H)

error_vec = Vector{Float64}()
time_vec = Vector{Float64}()
start_time = now()

W_grad = zeros(K)
H_grad = zeros(K)

@time for iteration = 1:num_iterations
    x_idx = rating[1][1]
y_idx = rating[1][2]
rv = rating[2]

    W_row = @view W[:, x_idx]
    H_row = @view H[:, y_idx]
    pred = dot(W_row, H_row)
    diff = rv - pred
    W_grad .= -2 * diff .* H_row
    H_grad .= -2 * diff .* W_row
    W[:, x_idx] .= W_row .- step_size .* W_grad
    H[:, y_idx] .= H_row .- step_size .* H_grad
end

@time if iteration % 4 == 1 ||
    iteration == num_iterations
    println("evaluate model")
    Orion.@parallel_for for rating in ratings
        x_idx = rating[1][1]
y_idx = rating[1][2]
rv = rating[2]

        W_row = @view W[:, x_idx]
        H_row = @view H[:, y_idx]
        pred = dot(W_row, H_row)
        err += (rv - pred)^2
    end
end
err = Orion.get_aggregated_value(:err, +)
curr_time = now()
elapsed = Int(Dates.value(curr_time - start_time)) / 1000
Orion.reset_accumulator(:err)
push!(error_vec, err)
push!(time_vec, elapsed)
end
println(error_vec)
println(time_vec)
Orion.stop()
exit()

A.2 Sparse Logistic Regression

include("/path/to/orion/src/julia/orion.jl")
Orion.set_lib_path("/path/to/orion/lib/liborion_driver.so")

const master_ip = "127.0.0.1"
const master_port = 10000
const comm_buff_capacity = 1024
const num_executors = 16
const num_servers = 16
Orion.glog_init()
Orion.init(master_ip, master_port, comm_buff_capacity, num_executors, num_servers)

const data_path = "file:///proj/BigLearning/jinlianw/data/kdda"
const num_iterations = 64
const step_size = Float32(0.00001)
const num_features = 20216830

Orion.@accumulator err = Float32(0)
Orion.@accumulator loss = Float32(0)
Orion.@accumulator line_cnt = 0

Orion.@share function parse_line(index::Int64, line::AbstractString)
global line_cnt += 1
tokens = split(strip(line), ' ')
label = parse(Int64, tokens[1])
if label == -1
    label = 0
    i = 1
    feature_vec = Vector{Tuple{Int64, Float32}}(length(tokens) - 1)
    for token in tokens[2:end]
        feature = split(token, ":")
        "feature["
feature_id = parse(Int64, feature[1])
@assert feature_id >= 1
feature_val = parse(Float32, feature[2])
feature_vec[i] = (feature_id, feature_val)
i += 1
return ((index,), (label, feature_vec))

Orion.@dist_array samples_mat = Orion.text_file(data_path, parse_line, is_dense = true, with_line_number = true, new_keys = true, num_dims = 1)
Orion.materialize(samples_mat)

line_cnt = Orion.get_aggregated_value(:line_cnt, :+)
println("number of lines read = ", line_cnt)

Orion.@dist_array weights = Orion.rand(num_features)
Orion.materialize(weights)

Orion.@share function sigmoid(z)
    return Float32(1.0) ./ (Float32(1.0) + exp(-z))
end

Orion.@share function safe_log(x)
    if abs(x) < Float32(1e-15)
        x = Float32(1e-15)
    end
    return log(x)
end

Orion.@dist_array weights_buf = Orion.create_sparse_dist_array_buffer((weights.dims ...), Float32(0.0))
Orion.materialize(weights_buf)

Orion.@share function apply_buffered_update(key, weight, update)
    return weight + update
end

Orion.set_write_buffer(weights_buf, weights, apply_buffered_update)

error_vec = Vector{Float32}()
loss_vec = Vector{Float32}()
time_vec = Vector{Float64}()
start_time = now()

for iteration = 1:num_iterations
    Orion.@parallel_for for sample in samples_mat
        sum = 0.0
        label = sample[2][1]
        features = sample[2][2]
        for feature in features
88     fid = feature[1]
89     fval = feature[2]
90     sum += weights[fid] * fval
91     end
92     diff = sigmoid(sum) - label
93     for feature in features
94         fid = feature[1]
95         fval = feature[2]
96             weights_buf[fid] -= step_size * fval * diff
97     end
98     end
99     if iteration % 1 == 0 ||
100         iteration == num_iterations
101             Orion.@parallel_for for sample in samples_mat
102                 sum = 0.0
103                 label = sample[2][1]
104                 features = sample[2][2]
105                 for feature in features
106                     fid = feature[1]
107                     fval = feature[2]
108                         sum += weights[fid] * fval
109                 end
110                 if label == 1
111                     loss += -safe_log(sigmoid(sum))
112                 else
113                     loss += -safe_log(1 - sigmoid(sum))
114                 end
115                 diff = sigmoid(sum) - label
116                 err += abs2(diff)
117             end
118             err = Orion.get_aggregated_value(:err , :+)
119             loss = Orion.get_aggregated_value(:loss , :+)
120             curr_time = now()
121             elapsed = Int(Dates.value(curr_time - start_time)) / 1000
122             println("iteration = ", iteration, " elapsed = ", elapsed, " err = ", err, " loss = ", loss)
123             push!(error_vec , err)
124             push!(loss_vec , loss)
125             push!(time_vec , elapsed)
126             Orion.reset_accumulator(:err)
127             Orion.reset_accumulator(:loss)
128         end
129     end
130
131 println(error_vec)
132 println(loss_vec)
133 println(time_vec)
134 Orion.stop()
Bibliography


2019-10-28. [Cited on page 16.]


[49] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth A. Gibson, and Eric P. Xing. High-performance distributed ML at scale through parameter server consis-


128


131
[99] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Tal-


133


[113] Chen Jin Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. Training deeper models by gpu memory optimization on tensorflow. 2017. [Cited on pages 16 and 89.]


[141] Richard Socher, Alex Perelygin, Jean Y. Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. [Cited on page 114.]


[175] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhit- 
ing Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An efficient com-
munication architecture for distributed deep learning on GPU clusters. CoRR, 
5.]

[176] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin 
Zheng. Exploring the hidden dimension in graph processing. In 12th USENIX 
Symposium on Operating Systems Design and Implementation (OSDI 16), pages 
1. URL https://www.usenix.org/conference/osdi16/technical-sessions/
presentation/zhang-mingxing. [Cited on page 77 and 77.]

models for regression and classification. In ICML, pages 1257–1264, 2009. [Cited on 
page 44.]

[178] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: 
A computation-centric distributed graph processing system. In 12th USENIX 
Symposium on Operating Systems Design and Implementation (OSDI 16), pages 
1. URL https://www.usenix.org/conference/osdi16/technical-sessions/
presentation/zhu. [Cited on page 77 and 77.]

[179] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. 
page 20.]

[180] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transfer-
able architectures for scalable image recognition. CoRR, abs/1707.07012, 2017. URL 