Machine Learning Parallelism Could Be Adaptive, Composable and Automated

Hao Zhang

September 2, 2020

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Greg R. Ganger
Jinyang Li (NYU)
Deva Ramanan
Christopher Ré (Stanford)
Eric P. Xing, Chair

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

Copyright © 2020 Hao Zhang.

The research in this thesis was jointly sponsored by: the National Science Foundation awards IIS-1447676 and CCF-1629559, the Defense Advanced Research Project Agency award FA872105C0003, and Petuum Inc. 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: Scalable Machine Learning, Machine Learning Parallelism, Distributed Deep Learning, Machine Learning System, Machine Learning for System, AutoML, Parameter Server, Composability, Neural Networks, Computer Vision
Abstract

In recent years, the pace of innovations in the fields of machine learning (ML) has accelerated. To cope with the sheer computational complexity of training large ML models on large datasets, researchers in SysML have created algorithms and systems that parallelize ML training and inference over multiple CPUs, GPUs, and computing nodes over a network. As ML models become more structurally complex, many systems have struggled to provide all-round performance on a wide variety of models – applying existing parallel training systems to complex models usually adds non-trivial development overheads in addition to ML prototyping, but often results in lower-than-expected performance.

In this thesis, we first present a simple design principle, adaptive parallelism, that applies suitable parallelization techniques to model building blocks (e.g. layers) according to their specific ML properties. Following it, we derive a series of new strategies and corresponding system implementations in different aspects of ML parallelization, that can adapt to the model and cluster specifications. We examine these strategies and systems, and show that they can significantly boost the efficiency or scalability of ML training on clusters from 2-10x in their applicable scenarios.

Generalized from these cases, we then present two novel and principled representations to express two classes of ML parallelisms: single-node dynamic batching and distributed ML parallelisms, and system architectures, Cavs and AutoDist, respectively. Cavs leverages a vertex-centric representation to address the substantial graph construction overheads for dynamic NNs, allows static optimizations which were only available to static NNs, and ease the parallel, batched computation of dynamic neural networks. AutoDist, on the other hand, introduces system-level composablealization, that can adapt to the model and cluster specifications. We examine these strategies and systems, and show that they can significantly boost the efficiency or scalability of ML training on clusters from 2-10x in their applicable scenarios.

Finally, we identify that ML scale-up is usually underestimated in terms of the amount of knowledge and time required to map from an appropriate distribution strategy to the model. On top of the composable representations and systems, we present an ML-based framework, AutoSync, to automatically optimize synchronization strategies in data-parallel distributed training. AutoSync navigates the space spanned by the proposed representation, and generates suitable distributed strategies for unseen models. We show that AutoSync can achieve high performance “out-of-the-box” – it automatically identifies synchronization strategies that report 1.2 - 1.6x speedups over existing hand-optimized systems, lowering the technical barrier of distributed ML and helping make it accessible to a larger community of users.
Acknowledgments

My advisor is cool. My thesis committee is nice.
# Contents

1 Introduction 1
   1.1 Motivating Example: Unsupervised Learning on Web-scale Text Data . . . . . . 1
   1.2 Thesis Overview ................................................................. 1
      1.2.1 Thesis Statement .......................................................... 1
      1.2.2 Thesis Outline, Contributions and Key Results ............................. 1

2 Background 5
   2.1 Machine Learning: A Computational Perspective .................................. 6
      2.1.1 The Master Equation ....................................................... 6
      2.1.2 Distributed Training ...................................................... 6
      2.1.3 Trends in Machine Learning ............................................. 6
   2.2 Hardware Infrastructures for Machine Learning .................................. 6
      2.2.1 Computing Hardware ...................................................... 6
      2.2.2 Cloud .......................................................... 6
      2.2.3 Trends in ML Hardware Infrastructure ................................ 6
   2.3 Machine Learning “Frameworks” .............................................. 6
      2.3.1 Trends in ML Frameworks .............................................. 6
   2.4 Distributed Systems for Large-scale Machine Learning ......................... 6
      2.4.1 Performance-improving Aspects in ML Parallelization .................. 6
      2.4.2 Writing Distributed Parallel ML Code .................................. 6
      2.4.3 Distributed System Autotuning ...................................... 6
      2.4.4 Notable Distributed ML Training Systems ............................... 6
      2.4.5 Trends in Distributed ML Systems ................................... 6
   2.5 Focused Problems of the Thesis ............................................... 7
      2.5.1 Optimize Performance .................................................. 7
      2.5.2 Improve Representation, Compsability, and Programmability of ML Parallelisms ................................................................. 7
      2.5.3 Automate ML Parallelization ............................................ 7

I Aspects of ML Parallelization 8

3 Scheduling and Communication 11
   3.1 Background ................................................................. 12
# Table of Contents

3.1.1 Distributed Training of Neural Networks ........................................... 12  
3.1.2 Communication Architectures .......................................................... 13  
3.1.3 Communication Challenges on GPU Clusters: An Example .................... 14  
3.2 Scheduling ............................................................................................... 16  
3.2.1 The Sequential Structure of DL Programs ........................................... 16  
3.2.2 Wait-free Backpropagation ................................................................ 16  
3.3 Adaptive Communication ....................................................................... 17  
3.4 Poseidon: An Efficient Communication Architecture for Distributed DL on GPU Clusters ................................................................. 19  
3.4.1 System Architecture ............................................................................ 19  
3.4.2 Integrate Poseidon with DL Frameworks ............................................. 22  
3.5 Evaluation ................................................................................................. 25  
3.5.1 Experiment Setup ................................................................................ 25  
3.5.2 Scalability ............................................................................................ 26  
3.5.3 Bandwidth Experiments ...................................................................... 31  
3.5.4 Comparisons to Other Methods ........................................................... 33  
3.5.5 Application: Scaling Up Image Classification on ImageNet 22K .......... 34  
3.6 Additional Related Work ......................................................................... 35  

4 Memory Management .................................................................................. 38  
4.1 Introduction ............................................................................................... 39  
4.1.1 Deep Learning on GPUs: a Memory Management Perspective ............ 39  
4.1.2 Memory Challenges for Distributed DL on GPU Clusters .................... 40  
4.2 Memory Swapping .................................................................................... 41  
4.2.1 Maintaining the Parameter Cache in GPU memory ............................. 43  
4.2.2 Pre-built Indexes and Batch Operations ............................................. 43  
4.2.3 Managing Limited GPU Device Memory ........................................... 44  
4.3 GeePS: Memory-optimized Parameter Server on GPUs .......................... 46  
4.3.1 GeePS Data Model and API ................................................................. 46  
4.3.2 Architecture ....................................................................................... 47  
4.3.3 Parallelizing Batched Access .............................................................. 48  
4.3.4 GPU Memory Management ............................................................... 49  
4.4 Evaluation ................................................................................................ 51  
4.4.1 Experimental Setup ............................................................................ 51  
4.4.2 Scaling Deep Learning with GeePS ...................................................... 54  
4.4.3 Dealing with Limited GPU Memory .................................................... 58  
4.5 Application: Hierarchical Deep Convolutional Neural Networks (HD-CNN) ........................................................................................ 61  
4.6 Additional Related Work ......................................................................... 62  

5 Consistency Model ....................................................................................... 64  
5.1 Distributed ML: Synchronous or Asynchronous? ...................................... 64  
5.2 Method and Setup .................................................................................... 66  
5.2.1 Simulation Model .............................................................................. 66  
5.2.2 Models and Algorithms ..................................................................... 66
II Composability and Representations of ML Parallelisms

6 Dynamic Neural Networks Parallelization

6.1 Dynamic Neural Networks
   6.1.1 Recurrent and Recursive Neural Networks (RNNs)

6.2 Programming and Parallelizing Dynamic Neural Networks
   6.2.1 Static Declaration
   6.2.2 Dynamic Declaration

6.3 DyNet: Dynamic Neural Network Toolkit
   6.3.1 DyNet Design Overview
   6.3.2 Efficient Graph Construction
   6.3.3 On-the-fly Dynamic Batching

6.4 Vertex-centric Representation
   6.4.1 Vertex-centric Programming Model
   6.4.2 Programming Interface

6.5 Cavs: An Efficient Runtime System for Dynamic Neural Networks
   6.5.1 Scheduling
   6.5.2 Memory Management
   6.5.3 Auto-differentiation
   6.5.4 Optimizing Execution Engine
   6.5.5 Implementation

6.6 Evaluation
   6.6.1 Experiment Setup
   6.6.2 Overall Performance
   6.6.3 Graph Construction
   6.6.4 Graph Optimizations

6.7 Additional Related Work

7 Composable Distributed Parallelization

7.1 Introduction
7.2 Overview

7.3 Synchronization Strategy Representation
   7.3.1 Graph-level Representation
   7.3.2 Node-level Representation
   7.3.3 Expressiveness

7.4 AutoDist: Composable System via Graph Rewriting
7.4.1 Strategy Verification ........................................... 118
7.4.2 Graph Rewriting Kernels ..................................... 118
7.4.3 Parallel Graph Rewriting ..................................... 119
7.4.4 Implementations ............................................... 119
7.5 Automatic Strategy Optimization ............................... 120
7.6 Evaluation ......................................................... 120
  7.6.1 Experiment Setup ............................................. 120
  7.6.2 Study of Synchronization Aspects .......................... 122
  7.6.3 Hand-composed Strategy Performance ..................... 125
7.7 Additional Related Work ......................................... 128

III Automatic ML Parallelization ................................. 129

8 Automatic Synchronization Strategy Optimization .......... 131
  8.1 Introduction ....................................................... 131
  8.2 Problem Formulation ............................................ 132
    8.2.1 Notations .................................................. 132
    8.2.2 Formulation ............................................... 132
    8.2.3 Search Space .............................................. 133
  8.3 AutoSync: Learning to Synchronize .......................... 134
    8.3.1 Domain Adaptive Simulator ............................... 134
    8.3.2 Knowledge-guided Search ................................ 137
    8.3.3 Low-cost Optimization using Transfer Learning ....... 140
  8.4 Evaluation ....................................................... 140
    8.4.1 Setup ...................................................... 140
    8.4.2 End-to-end Results and Ablation Studies ............... 142
    8.4.3 Transferring Trained Simulators ......................... 146
  8.5 Additional Related Work ....................................... 149

9 Conclusion and Future Work .................................... 150
  9.1 Conclusion ...................................................... 150
  9.2 Future Directions ............................................. 150

Bibliography ......................................................... 151
List of Figures

3.1 A convolutional neural network with 6 layers. ........................................... 12
3.2 An illustration of (a) the parameter server and (b) sufficient factor broadcasting for distributed ML. ................................................................. 14
3.3 (Top) Traditional backpropagation and (Bottom) wait-free backpropagation on distributed environment. ......................................................... 16
3.4 Comparisons of the three communication strategies when training AlexNet on GPU clusters. The parameters needed to be communicated between fc6 and fc7 are compared by varying (1) the number of cluster nodes $P$ and (2) batch size $K$. 19
3.5 An overview of Poseidon. The diagram shows the components added and managed by Poseidon on top of the original training process (the neural network on the left). ................................................................. 20
3.6 Throughput scaling when training GoogLeNet, VGG19 and VGG19-22K using Poseidon-parallelized Caffe and 40GbE bandwidth. Single-node Caffe is set as baseline (i.e., speedup = 1). ......................................................... 27
3.7 Throughput scaling when training Inception-V3, VGG19 and VGG19-22K using Poseidon-parallelized TensorFlow and 40GbE bandwidth. Single-node TensorFlow is set as baseline (i.e., speedup = 1). ......................................................... 29
3.8 Breakdown of GPU computation and stall time when training the three networks on 8 nodes using different systems. ......................................................... 30
3.9 (a) Speedup vs. number of nodes and (b) Top-1 test error vs. epochs for training ResNet-152 using Poseidon-TensorFlow and the original TensorFlow. ................. 31
3.10 Averaged communication load when training VGG19 using $TF$-$WFBP$, $Adam$ and $Poseidon$ with TensorFlow engine. Each bar represents the network traffic on a node. ................................................................. 31
3.11 Throughput scaling when training GoogLeNet, VGG19 and VGG19-22K using Poseidon-parallelized Caffe with varying network bandwidth. Single-node Caffe is set as baseline (speedup = 1). ......................................................... 32
3.12 Training loss and test error vs. iteration when training CIFAR-10 quick network using $Poseidon$ and $Poseidon-1bit$ on 4 GPUs with Caffe engine. ................. 34
4.1 A machine with a GPU device. ................................................................. 39
4.2 Single GPU deep learning and memory usage on contemporary ML frameworks. ................. 40
4.3 Parallel ML with parameter server. Parameter cache or data usually locate on shared CPU memory (DRAM). ................................................................. 41
4.4 Distributed ML on GPUs using a CPU-based parameter server. The right side of the picture is much like the single-GPU illustration in Figure 4.2. But, a parameter server shard and client-side parameter cache are added to the CPU memory, and the parameter data originally only in the GPU memory is replaced in GPU memory by a local working copy of the parameter data. Parameter updates must be moved between CPU memory and GPU memory, in both directions, which requires an additional application-level staging area since the CPU-based parameter server is unaware of the separate memories. ........................................... 42

4.5 Parameter cache in GPU memory. In addition to the movement of the parameter cache box from CPU memory to GPU memory, this illustration differs from Figure 4.4 in that the associated staging memory is now inside the parameter server library. It is used for staging updates between the network and the parameter cache, rather than between the parameter cache and the GPU portion of the application. ................................................................. 44

4.6 Parameter cache and local data partitioned across CPU and GPU memories. When all parameter and local data (input data and intermediate states) cannot fit within GPU memory, our parameter server can use CPU memory to hold the excess. Whatever amount fits can be pinned in GPU memory, while the remainder is transferred to and from buffers that the application can use, as needed. ... 46

4.7 Image classification throughput scalability for (upper) AdamLike model on ImageNet22K dataset and (bottom) GoogLeNet model on ILSVRC12 dataset. Both GeePS and CPU-PS run in the fully synchronous mode. ....................... 55

4.8 Image classification top-1 accuracies for (upper) AdamLike model on ImageNet22K dataset and (bottom) GoogLeNet model on ILSVRC12 dataset. ....................... 56

4.9 Video classification task: (upper) training throughput; (bottom) top-1 accuracies. 57

4.10 Per-layer memory usage of AdamLike model on ImageNet22K dataset. .......... 59

4.11 Throughput of AdamLike model on ImageNet22K dataset with different GPU memory budgets. ................................................................. 59

4.12 Training throughput on very large models. Note that the number of connections increases linearly with model size, so the per-image training time grows with model size because the per-connection training time stays relatively constant. ... 60

4.13 (a) A two-level category hierarchy taken from ImageNet-1000 dataset. (b) The hierarchical deep convolutional neural network (HD-CNN) architecture, where orange and yellow blocks are layers introduced in addition to the backbone model. 61

5.1 (a)(e) The number of batches to reach 71% test accuracy on CIFAR10 for 4 variants of ResNet with varying staleness, using 8 workers and SGD (learning rate 0.01) and Adam (learning rate 0.001). The mean and standard deviation are calculated over 3 randomized runs. (b)(d) The same metrics as (a)(c), but each model is normalized by the value under staleness 0 (s = 0), respectively. (e)(f) The number of batches to reach 92% accuracy for MLR and DNN with varying depths, normalized by the value under staleness 0. MLR with SGD does not converge within the experiment horizon (77824 batches) and thus is omitted in (f). 68
5.2 (a)(b)(c) The number of batches to reach 71% test accuracy on 1, 8, 16 workers with staleness \( s = 0, \ldots, 16 \) using ResNet8. We consider 5 variants of SGD: SGD, Adam, Momentum, RMSProp, and Adagrad. For each staleness level, algorithm, and the number of workers, we choose the learning rate with the fastest time to 71% accuracy from \{0.001, 0.01, 0.1\}. (d)(e)(f) show the same metric but each algorithm is normalized by the value under staleness 0 \( (s = 0) \), respectively, with possibly different learning rate.

5.3 (a) The number of batches to reach training loss of 0.5 for Matrix Factorization (MF). (b) shows the same metric in (a) but normalized by the values of staleness 0 of each worker setting, respectively (4 and 8 workers). (c)(d) Convergence of LDA log likelihood using 10 and 100 topics under staleness levels \( s = 0, \ldots, 20 \), with 2 and 16 workers. The convergence is recorded against the number of documents processed by Gibbs sampling. The shaded regions are 1 standard deviation around the means (solid lines) based on 5 randomized runs. (e)(f) The number of batches to reach test loss 130 by Variational Autoencoders (VAEs) on 1 worker, under staleness \( s = 0, \ldots, 16 \). We consider VAEs with depth 1, 2, and 3 (the number of layers in the encoder and the decoder networks, separately). The numbers of batches are normalized by \( s = 0 \) for each VAE depth, respectively. Configurations that do not converge to the desired test loss are omitted in the graph, such as Adam optimization for VAE with depth 3 and \( s = 16 \).

6.1 An example of a dynamic NN: (a) a constituency parsing tree, (b) the corresponding Tree-LSTM network. We use the following abbreviations in (a): S for sentence, N for noun, VP for verb phrase, NP for noun phrase, D for determiner, and V for verb.

6.2 A cell function shown in (a) could be applied on different structures such as a (b) chain (c) tree, or (d) graph.

6.3 The workflows of static declaration. Notations: \( D \) notates both the dataflow graph itself and the computational function implied by it; \( p \) is the index of a batch while \( k \) is the index of a sample in the batch.

6.4 The workflow of dynamic declaration.

6.5 The workflows of the vertex-centric programming model.

6.6 The vertex-centric representation expresses a dynamic model as a dynamic input graph \( G \) (left) and a static vertex function \( F \) (right).

6.7 The vertex function of an N-ary child-sum TreeLSTM [140] in Cavs. Within \( F \), users declare a computational dataflow graph using symbolic operators. The defined \( F \) will be evaluated on each vertex of \( G \) following graph dependencies.

6.8 Dynamic tensor.
6.9 The memory management at the forward pass of $\mathcal{F}$ (top-left) over two input trees (bottom-left). Cavs first analyzes $\mathcal{F}$ and inputs – it creates four dynamic tensors $\{\alpha_n\}_{n=0}^{3}$, and figures out there will be four batch tasks (dash-lined boxes). Starting from the first task (orange vertices $\{0, 1, 2, 5, 6, 7, 8, 9\}$), Cavs performs batched evaluation of each expression in $\mathcal{F}$. For example, for the pull expression $\alpha_0 = \text{pull}()$, it indexes the content of $\alpha_0$ on all vertices from the pull buffer using their IDs, and copies them to $\alpha_0$ continuously; for scatter and push expressions, it scatters a copy of the output ($\alpha_3$) to the gather buffer, and pushes them to the push buffer, respectively. Cavs then proceeds to the next batching task (blue vertices). At this task, Cavs evaluates each expression of $\mathcal{F}$ once again for vertices $\{3, 10, 11\}$. (e.g. for a pull expression $\alpha_0 = \text{pull}()$, it pulls the content of $\alpha_0$ from pull buffer again; for a gather expression $\alpha_2 = \text{gather}(1)$ at vertex 3, it gathers the output of the second child of 3, which is 1); it writes results continuously at the end of each dynamic tensor. It proceeds until all batching tasks are finished. 93

6.10 The dataflow graph encoded by $\mathcal{F}$ of Tree-LSTM. 94

6.11 Comparing five systems on the averaged time to finish one epoch of training on four models: Fixed-LSTM, Var-LSTM, Tree-FC and Tree-LSTM. In (a)-(d) we fix the hidden size $h$ and vary the batch size $bs$, while in (e)-(h) we fix $bs$ and vary $h$. 100

6.12 The averaged graph construction overhead per epoch when training (a) Tree-FC with different size of input graphs (b) Tree-LSTM with different batch size. The curves show absolute time in second (left y-axis), and the bar graphs show its percentage of the overall time (right y-axis). 101

6.13 Improvement of each optimization strategy on execution engine over a baseline configuration (speedup = 1). 103

7.1 How (a) prior DL training systems, and (b) a composable system differ in parallelizing an ML model. 110

7.2 An example of how the synchronization strategy is composed based on the DL model $\mathcal{G}$ and resource information in $\mathcal{R}$. 112

7.3 A composed strategy ($\S$7.3) and its mapped graph-rewriting operations. A solid arrow is from a tensor producer to a stateless consumer; dotted refers to a stateful variable update. (b) It shards variable $W$ into $W_0$, $W_1$ along axis 0. (c) It replicates onto two devices. (d) $W_0$ is synchronized by sharing the variable at one device and applying (Reduce, Broadcast), $W_1$ is synchronized with AllReduce. 114

7.4 An code snippet illustrating how to use AutoDist to distribute single-node ML code. 120

7.5 Comparing synchronization architectures across a diverse set of models and cluster specifications. PS1: load balanced PS builder without partitioning, PS2: load balanced PS builder with partitioning. AR: collective AllReduce based builder. Per-iter training time is reported. 123
7.6 Using a `AllReduce(chunk: int)` builder, we study how the performance is impacted by the collective merge scheme on different models and resource specifications. X-axis lists the value of chunk, the per-iter training time (lower is better) is reported. .............................. 124

7.7 Based on TF distributed runtime, we comparing `MirrorStrategy`, `TF-PS-LB`, `Horovod` with manually implemented `AllReduce` and `PS` builders on `AutoDist`. See §7.6.3 for a detailed analysis. .............................. 127

8.1 **Left:** Learning to synchronize framework. Initially, the simulator utilizes domain-agnostic features (§8.3.1) to explicitly estimate the runtime so to select promising strategies for evaluation (the blue line). After trials, the real runtime data are feedbacked to train the ML-based simulator to adapt to specific $G$, $D$, enhancing its capability in differentiating high-quality strategies. Gradually, the ML-based simulator takes over and directs the search (the red line). **Right:** Illustrations of the RNN and GAT simulators. .............................. 139

8.2 Comparing `AutoSync`, `AutoSync (-s)`, `AutoSync (-s,-k)` on (left axis) the improvement (higher is better) of the best found strategy over baseline, (right axis) the percentage of strategies better than baseline (higher is better), w.r.t. the number of trials conducted in 200 trials. The baseline (1x) is the better one of `PS` and `Horovod`). The average over 3 runs is reported. A curve is skipped from the plot if it is too below the baseline. .............................. 145

8.3 Transferring trained simulators from different source domains to 3 target domains, compared to untransferred `AutoSync-` and `AutoSync` with a budget of 100 trials. The average of three runs is reported. .............................. 148
List of Tables

3.1 The maximum throughput that commonly used Ethernet can provide in terms of how many Gigabits, Megabytes and number of float parameters could be transferred per second. ................................................................. 15

3.2 Statistics of modern CNN training, including the batch size, number of model parameters, per-iteration computation time and number of gradients generated per second on a single device. The performance is evaluated on a K40 GPU with standard hyperparameters in 2016. At the time this thesis is being written (2020), GPU FLOPS have been improved (e.g. Nvidia A100 [1]) with yet another order of magnitude (see §??), which consequently results in much heavier communication loads. ................................................................. 15

3.3 Parameter and FLOP distributions of convolution and fully-connected layers in AlexNet [82] and VGG-16 [130]. .................................................................................................................. 17

3.4 Estimated communication cost of PS, SFB and Adam for synchronizing the parameters of a $M \times N$ FC layer on a cluster with $P_1$ workers and $P_2$ servers, when batchsize is $K$. ................................................................. 18

3.5 Poseidon APIs for parameter synchronization. ............................................. 24

3.6 Neural networks for evaluation. Single-node batchsize is reported. The batchsize is chosen based on the standards reported in literature (usually the maximum batch size that can fill in the GPU memory is used). .................................................. 27

3.7 Comparisons of the image classification results on ImageNet 22K. ................ 34

4.1 Model training configurations. ........................................................................ 54

4.2 Errors on ImageNet validation set. ................................................................. 62

5.1 Overview of the models, algorithms [34, 49, 77, 121? ], and dataset in our study. $\eta$ denotes learning rate, which, if not specified, are tuned empirically for each algorithm and staleness level (over $\eta = 0.001, 0.01, 0.1$), $\beta_1, \beta_2$ are optimization hyperparameters (using common default values). $\alpha, \beta$ in LDA are Dirichlet priors for document topic and word topic random variables, respectively. ................................................................. 67

6.1 A side-by-side comparison of existing programming models for dynamic NNs, and their advantages and disadvantages. ................................................................. 88

6.2 The averaged computation time (Cavs/Fold/DyNet) and the speedup (Cavs vs Fold/DyNet) for training one epoch on Tree-FC with varying size of the input trees (left part), and on Tree-LSTM with varying batch size (right part). . 102
6.3 Breakdowns of average time per epoch on memory-related operations and computation, comparing Cavs to DyNet on training and inference of Tree-LSTM with varying $bs$. ................................................................. 103

7.1 The proposed strategy space in AutoSync covers many advanced strategies in existing systems. Moreover, it offers a superset of semantics beyond them. ....... 117

7.2 Cluster specifications we have experimented with, listed with their reference name ($\mathcal{R}$), setup information, GPU distributions, and bandwidth specifications. .... 121

7.3 ML models we have experimented with. All the model implementations are from the tensorflow/models repository. IC: image classification, MT: machine translation, LM: language modeling. For neural collaborative filter (NCF), we follow MLPerf [97] and use a enlarged version (x16x32), and we use dense instead of sparse gradients for its embedding variables to test systems’ capability. ................................................................. 122

8.1 Comparisons of different model instantiations of the simulator. ..................... 142

8.2 The per-iteration time statistics of 200 strategies found by RS and GA on (VGG16, A). ................................................................. 142

8.3 Studies on feature importance (pairwise ranking accuracy is reported). ........... 144

8.4 The target domain test accuracy under different transfer learning settings. ....... 146
Chapter 1

Introduction

1.1 Motivating Example: Unsupervised Learning on Web-scale Text Data

1.2 Thesis Overview

1.2.1 Thesis Statement

In this thesis, we describe a set of ML and system techniques that together support the following statement:

*The structured and composable nature of nowadays ML programs allows for building optimized parallelization strategies that are adaptive to model and cluster specifications, composable from different ML parallelisms, and auto-generated via learning-based methods.*

1.2.2 Thesis Outline, Contributions and Key Results

This thesis statement is supported with three major research components, that map to the three parts of this thesis, with the detailed contributions and key results elaborated below.

Understanding and Optimize ML Parallelization with Adaptive Parallelisms

Different ML model structures, statistical properties of their building components, and hardware environment could impact the effectiveness of almost all ML parallelization technologies, such as synchronization architectures, consistency models, shared memory management, etc. We develop a principled methodology called *adaptive parallelism* – that factors the parallelization strategy with respect to the model. Based on the structure of ML programs, it adapts to specific statistical properties and computational patterns of the models to be distributed. We demonstrate that adaptive parallelisms can boost the scalability for 5 - 10x across multiple models, environment and task workloads, and open up the design space for models with more complexity and
more hierarchical architectures, which would otherwise be restricted by hardware limitations or computational costs. This part of research leads to the following contributions:

- It introduces an adaptive scheduling algorithm, wait-free backpropagation (WFBP) (§3.2), and a hybrid communication architecture (§3.3). Based on the two key techniques we build the Poseidon system, one of the state-of-the-art distributed DL systems for CNNs.

- It presents a shared memory management mechanism, memory swapping (Chapter 4), specialized to address the memory limitations in distributed deep learning on GPU clusters.

- Through empirical studies on how staleness impacts the training of various models and deep learning building blocks (Chapter 5), it reveals that the consistency model should also flexibly adapt to model specifications to best trade-off between system throughput and statistical efficiency.

- A system-ML co-designed model, HD-CNN (§4.5), is presented. With increased model complexity and training scale, it achieved state-of-the-art image classification results on ImageNet.

Representations of Parallelisms

We generalize from lessons learned in this first part – conditioned on the representation of nowadays ML models, we develop ways of expressing a rich set of ML parallelisms, originated from different aspects (synchronization architecture, node/layer partitioning and placement, consistency control, gradient compression, etc.), within a unified representation. The formulations offer a few notable advantages: based on the representations, we are able to compose parallelization strategies using a few atomic parallelization aspects as building blocks; they simplify parallel ML programming for DL models, isolate the sophistication of distributed systems from the ML prototyping process; it allows joint optimization of multiple performance-improving factors and strategy optimization at sub-model granularity. This part of research contains the following contributions:

- It contributes to an open source DL framework, DyNet (§6.3), specialized for dynamic neural networks based on dynamic declaration (§6.2.2).

- One step further, it introduces a new vertex-centric representation for dynamic neural networks (§6.4). The representation derives a vertex-centric programming model (§6.4.2), a parallelization strategy for dynamic batching, and allows static dataflow graph optimization techniques to be utilized in dynamic neural network training. Based on these techniques, we implement the system Cavs for dynamic neural network. Cavs achieves state-of-the-art training performance on a variety of neural networks with dynamic structures.

- It formalizes a unified representation of distribution strategies for static DL models (§7.3). This representation allows parallelization strategies to only depend on model and resource specifications, and to be composed from base parallelisms. Based on this representation, we develop AutoDist (§7.4), a composable system for simplified and faster programming for ML parallelization, for unseen models, with matched or even better performance than
specialized systems.

**Automating ML Parallelization**

Backed by the developed representations and system implementations, we develop end-to-end pipeline to automatically generate synchronization strategies so as to automate ML parallelization. This part of the research contains the following contributions:

- It constructs a novel search space of synchronization-affecting factors and learns to optimize synchronization strategies on variable-level towards better training performance (Chapter 8).

- AutoSync is able to find synchronization strategies up to 1.6x better than those manually optimized, on a variety of models, even with only 200 trial run budget.

- It develops transfer-learning mechanisms to further reduce the auto-optimization cost – AutoSync can transfer among similar model architectures, among similar cluster configurations, or both.

- A dataset containing 10000 strategy, model, cluster, and runtime tuples is made available to facilitate future research in automating the parallelization of ML programs.
Chapter 2

Background

2.1 Machine Learning: A Computational Perspective

2.1.1 The Master Equation
2.1.2 Distributed Training
2.1.3 Trends in Machine Learning
   More Complexity
   More Composability
   More Automation

2.2 Hardware Infrastructures for Machine Learning

2.2.1 Computing Hardware
2.2.2 Cloud
2.2.3 Trends in ML Hardware Infrastructure

2.3 Machine Learning “Frameworks”

2.3.1 Trends in ML Frameworks

2.4 Distributed Systems for Large-scale Machine Learning

2.4.1 Performance-improving Aspects in ML Parallelization
   ML Properties vs. Choices of Parallelization Aspects
2.4.2 Writing Distributed Parallel ML Code
2.4.3 Distributed System Autotuning
2.4.4 Notable Distributed ML Training Systems
2.4.5 Trends in Distributed ML Systems
2.5 Focused Problems of the Thesis

2.5.1 Optimize Performance

2.5.2 Improve Representation, Compsability, and Programmability of ML Parallelisms

2.5.3 Automate ML Parallelization
Part I

Aspects of ML Parallelization
Overview

The first part of the thesis studies individual aspects involved with distributed ML, and/or derive optimizations on each aspect based on the suggested approach – adaptive parallelism.

Chapter 3 reveals communication as one of the core challenges in contemporary distributed machine learning. By exploiting the layered structure of deep neural networks, the chapter develops a scheduling approach, wait-free backpropagation, and an adaptive communication algorithm, to alleviate bursty network traffics and to reduce communication loads, respectively. Based on these methods, this chapter demonstrates a distributed ML system, Poseidon, that boosts the scalability of various DL models by 2x-10x on top of existing DL frameworks.

Chapter 4 identifies the data movement overheads, GPU stalls, and limited memory as three major hurdles on scaling big models on distributed GPUs. Once again, by adapting to the model structure, we develop the memory swapping technique as an effective way of training large models on limited GPU memory. Based on memory swapping, this chapter presents GeePS, a parameter server system specialized for scaling deep learning applications across GPUs. GeePS allows data-parallel training of very large neural networks, bounded by the largest layer rather than the overall model size – it is able to train DL models that would otherwise require 7x more memory (70GB) than a standard GPU normally offers (10GB).

Chapter 5 brings in the aspect of consistency model into the picture. It studies the impact of staleness, a quantitative characterization of the tolerance of delayed updates allowed in a distributed ML system, against various ML models, algorithms, and hyperparameter settings. Through simulation studies, this chapter reveals that different models or even model building blocks have different degrees of robustness against staleness. This suggest design space for future systems to smartly adapt the consistency to best trade-off between system throughput and statistical efficiency.

The results presented in this part of the thesis have appeared in the following publications:


- Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric Xing. GeePS:


Chapter 3

Scheduling and Communication

Nowadays ML models possess hundreds of millions of floating-point parameters and their training computation usually exhibits iterative-convergent patterns. The high computational cost of DL programs on large-scale data necessitates the training on a distributed GPU cluster in order to keep the training time acceptable.

ML software such as TensorFlow [3], PyTorch [38], MxNet [14], Caffe [68] allow practitioners to easily experiment with DL models on a single machine. However, their distributed implementations can scale poorly for larger models. For example, we find that on the VGG19-22K network [129] (229M parameters) or BERT-large [32] (), open-source TensorFlow on 32 machines can be slower than single machine (Section 3.5.2). This observation underlines the challenge of scaling DL on GPU clusters: the high computational throughput of GPUs allows more data batches to be processed per minute (than CPUs), leading to more frequent network synchronization that grows with the number of machines. Existing communication strategies, such as parameter servers (PS) for ML [87, 152], can be overwhelmed by the high volume of communication [23]. Moreover, despite the increasing availability of faster network interfaces such as Infiniband or 40GbE Ethernet, GPUs have continued to grow rapidly in computational power, and continued to produce parameter updates faster than can be naively synchronized over the network. For instance, on a 16-machine cluster with 40GbE Ethernet and one Titan X GPU per machine, updates from the VGG19-22K model will bottleneck the network, so that only an 8x speedup over a single machine is achieved (§ 3.5.2).

These scalability limitations in distributed DL stem from at least two causes: (1) the gradient updates to be communicated are very large matrices, which quickly saturate network bandwidth; (2) the iterative nature of DL algorithms causes the updates to be transmitted in bursts (at the end of an iteration or batch of data), with significant periods of low network usage in between.

Following the design principle mentioned in §??, in this chapter, we derive strategies to improve the performance in the scope of neural network training, specifically concerning the following two ML parallelization aspects: computation and communication scheduling (which we refer as scheduling in this chapter) and distributed communication.
3.1 Background

3.1.1 Distributed Training of Neural Networks

In this section, we first instantiate the generic formulation showed in §??? in the context of distributed training of neural networks on GPU clusters, so to reveal the specific ML or computational properties these models or training workloads possess. Based on that, we derive adaptive scheduling and communication strategies to fit with these properties.

Neural networks are a family of hierarchical models containing many layers, from as few as 5-10 [82] to as many as 100s [57]. Figure 3.1 illustrates a convolutional neural network with 6 layers. The first layer (green) is an input layer that reads data in application-specific formats, e.g., raw pixels if it is trained to classify images. The input layer is connected to a sequence of intermediate layers (cyan, orange), each of which consists of a few neurons, where each neuron applies a function transformation $f$ on its input and produces an output. A vector output is obtained by concatenating the output of all neurons from a layer. By stacking multiple intermediate layers, the NN can transform raw input data one layer at a time, first into a series of intermediate representations, and finally into the desired output or prediction (red). DL programmers usually need to specify the computation of a layer by defining two properties of its neurons. The first is the transformation function $f(W, x)$, where $x$ is the input to the neuron, and $W$ is an optional trainable parameter. The other is the connectivity that determines how the neuron should be connected to its adjacent layer. For instance, a convolutional neural network has two types of neuron: convolutional (CONV) neuron (cyan) that are only locally connected to a subset of neurons in its previous layer, and fully-connected (FC) neurons (orange).

![Figure 3.1: A convolutional neural network with 6 layers.](image)

Most NNs need to be trained with data to give accurate predictions. Stochastic gradient descent (SGD) and backpropagation are commonly employed to train NNs iteratively – each iteration performs a feed forward (FF) pass followed with a backpropagation (BP) pass. In the FF pass, the network takes a training sample as input, forwards from its input layer to output layer to produce a prediction. A loss function is defined to evaluate the prediction error, which is then backpropagated through the network in reverse, during which the network parameters are updated by their gradients towards where the error would decrease. After repeating a sufficient number of passes, the network will usually converge to some state where the loss is close to a minima, and the training is then terminated.

Starting from Equation ??, given data $D$ and a loss function $\mathcal{L}$, fitting the parameters $\theta$ of a
NN can be formulated as the following \textit{iterative-convergent} algorithm that repeatedly executing the update equation

\[
\theta^{(t)} = \theta^{(t-1)} + \epsilon \cdot \nabla \mathcal{L}(\theta^{(t-1)}, D^{(t)})
\]  

(3.1)

until \( \theta \) reaches some stopping criteria, where \( t \) denotes the iteration. The update function \( \nabla \mathcal{L} \) calculates the gradients of \( \mathcal{L} \) over current data \( D_i (D_i \in D) \). The gradients are then scaled by a learning rate \( \epsilon \) and applied on \( \theta \) as updates. As the gradients are additive over data samples \( i \), i.e. \( \theta^{(t)} = \theta^{(t-1)} + \epsilon \cdot \sum_i \nabla \mathcal{L}(\theta^{(t-1)}, D_i) \), for efficiency, we usually feed a batch of training samples \( D^{(t)} (D^{(t)} \subset D) \) at each training iteration \( t \), as in Equation 3.1.

In large-scale training, data \( D \) are usually too large to process on a single machine in acceptable time. To speedup the training, we usually resort to \textit{data parallelism} (§??), a parallelization strategy that partitions the data \( D \) and distributes to a cluster of computational worker machines (indexed by \( p = 1, \cdots, P \)), as illustrated in Figure 3.2. At each iteration \( t \), every worker fetches a batch \( D_p^{(t)} \) from its data partition and computes the gradients \( \nabla \mathcal{L}(\theta^{(t)}, D_p^{(t)}) \). Gradients from all workers are then aggregated and applied to update \( \theta^{(t)} \) to \( \theta^{(t+1)} \) following

\[
\theta^{(t+1)} = \theta^{(t)} + \epsilon \sum_{p=1}^{P} \nabla \mathcal{L}(\theta^{(t)}, D_p^{(t)})
\]  

(3.2)

Data-parallelism allows data to be locally partitioned to each worker, which is advantageous for large datasets. It however requires every worker to have read and write access to the shared model parameters \( \theta \), which causes communication among workers; this communication support can be provided by a parameter server architecture [19, 152] (Figure 3.2a) or a peer-to-peer broadcasting architecture [155] (Figure 3.2b), both were originally designed for general-purpose data-parallel ML programs on CPUs.

### 3.1.2 Communication Architectures

We briefly introduce how to use a Parameter Server or a Sufficient Factor Broadcasting architecture to provide the required communication support in the distributed neural network training.

**Parameter Server**

A parameter server (PS) is a distributed shared memory system that provides systematic abstraction of iterative-convergent algorithms in data-parallel distributed ML. Typically, PS enables each worker to access the global model parameters \( \theta \) via network communications following the client-server scheme. Neural network training can be trivially parallelized over distributed workers using PS with the following 3 steps: (1) Each worker computes the gradients (\( \nabla \mathcal{L} \)) on their own data partition and send them to remote servers; (2) servers receive the updates and apply (+) them on globally shared parameters; (3) a consistency scheme coordinates the synchronization among servers and workers (Figure 3.2a).
Sufficient Factor Broadcasting

Many ML models represent their parameters \( \theta \) as matrices. For example, fully-connected NNs, when trained using SGD, their gradient \( \nabla \theta \) over a training sample is a rank-1 matrix, which can be cast as the outer product of two vectors \( u, v \): \( \nabla \theta = uv^\top \), where \( u \) and \( v \) are called sufficient factors (SFs). Hence, Sufficient factor broadcasting (SFB) [155] can be employed to parallelize these models by broadcasting SFs among workers and then reconstructing the gradient matrices \( \nabla \theta \) using \( u, v \) locally. SFB presents three key differences from PS: (1) SFB uses a P2P communication strategy that transmits SFs instead of full matrices. (2) Unlike gradients, SFs are not additive over training samples, i.e., the number of SFs needed to be transmitted grows linearly with the number of data samples (not data batches); (3) the overall communication overheads of SFB increase quadratically with the number of workers.

3.1.3 Communication Challenges on GPU Clusters: An Example

Modern DL models are mostly trained using NVIDIA GPUs, because the primary computational steps (e.g., matrix-matrix multiplications) in DL match the SIMD operation that could be efficiently performed by GPUs. In practice, DL practitioners often use single-node software frameworks which mathematically derive the correct training algorithm and execute it on GPU by calling GPU-based acceleration libraries, such as CUBLAS and cuDNN. It is thus straightforward to parallelize these programs across distributed GPUs using either PS or SFB, by moving the computation from CPU to GPU, and performing memory copy operations (between DRAM and GPUs) or communication (among multiple nodes) whenever needed. However, we argue below and show empirically in §3.5 that these usually lead to suboptimal performance.

The inefficiency is mainly caused by parameter synchronization via the network. Compared to CPUs, GPUs are an order of magnitude more efficient in matrix computations; the production of gradients on GPUs is much faster than they can be naively synchronized over the network. As
<table>
<thead>
<tr>
<th>Ethernet</th>
<th>Rate (GBit/s)</th>
<th>Rate (Mb/s)</th>
<th>Rate (# floats/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 GbE</td>
<td>1</td>
<td>125</td>
<td>31.25M</td>
</tr>
<tr>
<td>10 GbE</td>
<td>10</td>
<td>1250</td>
<td>312.5M</td>
</tr>
<tr>
<td>Infiband</td>
<td>40</td>
<td>5000</td>
<td>1250M</td>
</tr>
<tr>
<td>100 GbE</td>
<td>100</td>
<td>12500</td>
<td>3125M</td>
</tr>
</tbody>
</table>

Table 3.1: The maximum throughput that commonly used Ethernet can provide in terms of how many Gigabits, Megabytes and number of float parameters could be transferred per second.

<table>
<thead>
<tr>
<th>Model</th>
<th>Batch size (# images)</th>
<th># Parameters (# floats)</th>
<th>Time (s/iter)</th>
<th>Gradients (# floats/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet</td>
<td>256</td>
<td>61.3M</td>
<td>0.96</td>
<td>63.85M</td>
</tr>
<tr>
<td>VGG-16</td>
<td>64</td>
<td>128.3M</td>
<td>4.06</td>
<td>31.60M</td>
</tr>
</tbody>
</table>

Table 3.2: Statistics of modern CNN training, including the batch size, number of model parameters, per-iteration computation time and number of gradients generated per second on a single device. The performance is evaluated on a K40 GPU with standard hyperparameters in 2016. At the time this thesis is being written (2020), GPU FLOPS have been improved (e.g. Nvidia A100 []) with yet another order of magnitude (see §??), which consequently results in much heavier communication loads.

As a result, the training computations are usually bottlenecked by communications.

Table 3.1 lists the standards of commonly used Ethernet and Table 3.2 shows some statistics of modern CNN training. To understand the statistics, take the training of AlexNet [82] (61.5M parameters) on Titan X with a standard batch size 256, 240 million gradients will be generated per second on each GPU (0.25s/batch). If we parallelize the training on 8 nodes using a PS, with every node also holding 1/8 of parameters as a PS shard; then, every node needs to transfer $240M \times 7/8 \times 4 = 840M$ float parameters in one second to make sure the next iteration of computation not being blocked. Apparently, the demanded throughput (>26Gbps) exceeds the bandwidth that commodity Ethernet (i.e., 1GbE and 10GbE Ethernet) provides; the GPUs distributed across clusters cannot be fully utilized. Practically, it is usually difficult to partition the parameters completely equally, which will result in more severe bandwidth demands, or bursty communication traffic on several server nodes (as we will show in §3.5.4), which prevents the trivial realization of efficient DL on distributed GPUs. In addition, frequent memory copy operations between DRAM and GPU memory can cause non-trivial overheads.

We propose that a solution to these challenges should exploit the structure of DL algorithms to improve scheduling and communication: on one hand, it should identify ways in which the matrix updates can be separated from each other, and then schedule them in a way that avoids bursty network traffic. On the other hand, the solution should also exploit the structure of the matrix updates themselves, and wherever possible, reduce their size and thus the overall communication load on the network.
3.2 Scheduling

3.2.1 The Sequential Structure of DL Programs

At the core of the DL program is the BP algorithm that performs forward-backward pass through the network repeatedly. If we define a forward and a backward pass through the $l$th layer of a network as $f_l^t$ and $b_l^t$, respectively, then a Computation step at iteration $t$ is notated as $C_t = [f_1^t, \ldots, f_L^t, b_L^t, \ldots, b_1^t]$, as illustrated in Figure 3.3. When executing on distributed GPUs, inter-machine communications are required after each $C$ step to guarantee the synchronized replication of model parameters. We similarly define the Synchronization step $S_t$ as the process that a worker sends out locally generated updates and then receives updated parameters from remote workers at iteration $t$. Therefore, a naive parallelization of DL training over distributed GPUs using either PS or SFB can be expressed as alternating $C_t$ and $S_t$ defined above. We note that DL training is highly sequential; the communication and computation perform sequentially, waiting each other to finish (top of Figure 3.3).

We note that as every layer of a NN contains an independent set of parameters, $S_t$ can be decoupled as $S_t = (s_1^t, \ldots, s_L^t)$, by defining $s_l^t$ as the synchronization of parameters of layer $l$. If we further decompose $s_l^t = [o_l^t, i_l^t]$ as first sending out local updates of layer $l$ ($o_l^t$) and reads in the updated parameters remotely ($i_l^t$), we can rewrite a training iteration as: $[C_t, S_t] = [f_1^t, \ldots, f_L^t, b_L^t, \ldots, b_1^t, o_L^t, \ldots, o_1^t, i_L^t, \ldots, i_1^t]$. The sequential nature of the BP algorithm presents us an opportunity to overlap the computations and communications through rescheduling: We next present wait-free backpropagation to overlap $C_t$ and $S_t$ by partially rescheduling those $b_t$ and $s_t$ that are independent.

3.2.2 Wait-free Backpropagation

The wait-free backpropagation (WFBP) is designed to overlap communication overheads with the computation based on two key independencies in the program: (1) the send-out operation $o_l^t$
Table 3.3: Parameter and FLOP distributions of convolution and fully-connected layers in AlexNet [82] and VGG-16 [130].

<table>
<thead>
<tr>
<th></th>
<th>CONV Layers (#/#% )</th>
<th>FC Layers (#/#% )</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet</td>
<td>2.3M / 3.75</td>
<td>59M / 96.25</td>
</tr>
<tr>
<td>VGG-16</td>
<td>7.15M / 5.58</td>
<td>121.1M / 94.42</td>
</tr>
</tbody>
</table>

WFBP is most beneficial for training DL models that have their parameters concentrating at upper layers (FC layers) but computation concentrating at lower layers (CONV layers), e.g., VGG [130] and AdamNet [19, 23]), as illustrated in Table 3.3; it overlaps the communication of top layers (90% of communication time) with the computation of bottom layers (90% of computation time) [23, 167]. Besides chain-like NNs, WFBP is generally applicable to other non-chain like structures (e.g., tree-like structures), as the parameter optimization for deep neural networks depends on adjacent layers (and not the whole network), there is always an opportunity for parameter optimization (i.e., computation) and communication from different layers to be performed concurrently.

Some DL frameworks, such as TensorFlow, represent the data dependencies of DL programs using graphs, therefore implicitly enable auto-parallelization. However, they fail on exploring the potential opportunities of parallelization between iterations. For example, TensorFlow needs to fetch the updated parameters from the remote storage at the beginning of each iteration, while it is possible to overlap this communication procedure with the computation procedure of the previous iteration. In comparison, WFBP enforces this overlapping by explicitly pipelining compute, send and receive procedures. We describe our implementation of WFBP in Section 3.4.2 and empirically show its effectiveness in Section 3.5.2.

3.3 Adaptive Communication

While WFBP-based scheduling overlaps communication and computation, it does not reduce the communication overhead. In situations where the network bandwidth is limited (e.g., commodity Ethernet or the Ethernet is shared with other communication-heavy applications), the communication would still be unacceptably slow. To address the issue, we introduce a adaptive

---

1 Most classification models will fall into this family if the number of classes to be classified is large.
Table 3.4: Estimated communication cost of PS, SFB and Adam for synchronizing the parameters of a $M \times N$ FC layer on a cluster with $P_1$ workers and $P_2$ servers, when batchsize is $K$.

<table>
<thead>
<tr>
<th>Method</th>
<th>Server</th>
<th>Worker</th>
<th>Server &amp; Worker</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS</td>
<td>$2P_1MN/P_2$</td>
<td>$2MN$</td>
<td>$2MN(P_1+P_2-2)/P_2$</td>
</tr>
<tr>
<td>SFB</td>
<td>N/A</td>
<td>$2K(P_1-1)(M+N)$</td>
<td>N/A</td>
</tr>
<tr>
<td>Adam (max)</td>
<td>$P_1MN + P_1K(M+N)$</td>
<td>$K(M+N) + MN$</td>
<td>$(P_1-1)(MN + KM + KN)$</td>
</tr>
</tbody>
</table>

Our idea comes from two observations: the synchronization operations $\{S_l^i\}_{i=1}^L$ (§3.2.1) are independent of each other, meaning that we can use different communication methods for different $S_l^i$ by specializing $o_l^i$ and $i_l^i$ according to the two methods described in Figure 3.2; second, a NN structure is usually predefined and fixed throughout the training – by measuring the number of parameters needed to transferred, we are able to estimate the communication overhead, so that we can always choose the optimal method even before the communication happens.

Consider training VGG19 network [130], the overheads of $S_l^i$ could be estimated as follows (Table 3.4): assume the batch size $K = 32$, the number of workers and server nodes $P_1 = P_2 = 8$ (assume parameters are ideally equally partitioned over all server shards), respectively. On one hand, if $l$ is an FC layer (with shape $4096 \times 4096$, $M = N = 4096$), synchronizing its parameters via PS will transfer $2MN \approx 34$ million parameters for a worker node, $2P_1MN/P_2 \approx 34$ million for a server node, and $2MN(P_1+P_2-2)/P_2 \approx 58.7$ million for a node that is both a server and a worker, compared to $2K(M+N)(P_1-1) \approx 3.7$ million for a single node using SFB. On the other hand, if $l$ is a CONV layer, the updates are indecomposable and sparse, so we can directly resort to PS. Therefore, the synchronization overheads depend not only on the model (type, shape, size of the layer), but also the size of the clusters. The optimal solution usually changes with $M, N, K, P_1, P_2$. Determining the optimal communication architecture requires taking into account these factors, so to dynamically adjust the communication method for different parts of a model – it always chooses the best method from available ones whenever it results in fewer communication overheads.

Microsoft Adam [19] employs a different communication strategy from those in Figure 3.2. Instead of broadcasting SFs across workers, they first send SFs to a parameter server shard, then pull back the whole updated parameter matrices. This seems to reduce the total number of parameters needed to be communicated, but usually leads to load imbalance; the server node that holds the corresponding parameter shard overloads because it has to broadcast the parameter matrices to all workers ($P_1MN + P_1K(M+N)$ messages need to be broadcasted), which easily causes communication bottleneck (Section 3.5.4). It is noticeable that reconstructing gradients from SFs may cause extra computation cost, which however is often negligible compared to communication. We describe our implementation of adaptive communication in the next section, and assess its effectiveness in §3.5.
Figure 3.4: Comparisons of the three communication strategies when training AlexNet on GPU clusters. The parameters needed to be communicated between fc6 and fc7 are compared by varying (1) the number of cluster nodes \( P \) and (2) batch size \( K \).

3.4 Poseidon: An Efficient Communication Architecture for Distributed DL on GPU Clusters

Starting with the scheduling and communication strategies, we derive and implement Poseidon, an efficient communication architecture for data-parallel DL on distributed GPUs. Poseidon exploits the sequential layer-by-layer structure in DL programs, finding independent GPU computation operations and network communication operations in the training algorithm, so that they can be scheduled together to reduce bursty network communication. Moreover, Poseidon implements a hybrid communication scheme that accounts for each DL program layer’s mathematical properties as well as the cluster configuration, in order to compute the network cost of different communication methods, and select the cheapest one – currently, Poseidon implements and supports a parameter server scheme [152] that is well-suited to small matrices, and a sufficient factor broadcasting scheme [155] that performs well on large matrices. This section first elaborates Poseidon’s system architecture and APIs, and then describes how to modify a framework using Poseidon to enable distributed execution.

3.4.1 System Architecture

Figure 3.5 illustrates the architecture of Poseidon: a C++ communication library that manages parameter communication for DL programs running on distributed GPUs. It has three main components: coordinator, that maintains the model and the cluster configuration; KV store, a shared memory key-value store that provides support for parameter server based communication; client library, which is plugged into DL programs to handle parameter communication. Their APIs are listed in Table 3.5.
Figure 3.5: An overview of Poseidon. The diagram shows the components added and managed by Poseidon on top of the original training process (the neural network on the left).

**Coordinator**

To setup distributed training, the client program (e.g., Caffe) first instantiates Poseidon by creating a coordinator within its process. Coordinators will first collect necessary information, including the cluster information (e.g., the number of workers and server nodes, their IP addresses) and the model architecture (e.g., the number of layers, layer types, number of neurons and how they are connected, etc.). With the information, the coordinator will initialize the KV stores and the client library with two steps: (1) allocate proper communication ports for each PS shard and peer worker; (2) determine what parameters should be transmitted via the KV store and what by SFB, and hash the parameters equally to each KV store if necessary, and save the mapping in the information book, which, throughout the whole training, is maintained and synchronized across nodes, and could be accessed elsewhere through coordinator’s Query API. Besides, the coordinator provides another API BestScheme that takes in a layer and returns the optimal communication scheme for it according to the strategy described in §3.3 (Algorithm 1).

**KV Store**

The KV store is implemented based on a bulk synchronous parameter server [23, 152], and instantiated by coordinators on a list of user-specified “server” machines. Each instance of the KV store holds one shard of the globally shared model parameters in the form of a set of KV pairs, of which each KV pair is stored on a chunk of DRAM. Poseidon sets the size of a KV pair to a fixed small size (e.g., 2MB), so as to partition and distribute model parameters to server nodes.
Algorithm 1 Get the best comm method of layer $l$

1: function BESTSCHEME($l$)
2:     $layer\_property = \text{Query}(l\.name)$
3:     $P_1, P_2, K = \text{Query}(n\_worker, n\_server, batchsize)$
4:     if $layer\_property\.type == \text{FC}$ then
5:         $M = layer\_property\.width$
6:         $N = layer\_property\.height$
7:     if $2K(P_1 - 1)(M + N) \leq \frac{2MN(P_1 + P_2 - 2)}{P_2}$ then
8:         return SFB
9:     end if
10: end if
11: return PS
12: end function

as equally as possible, reducing the risk of Ethernet bottleneck. Each KV store instance manages a parameter buffer on RAM, and provides PS-like APIs, such as Receive and Send, for receiving and applying updates from client libraries, or sending out parameters. It will regularly checkpoint current parameter states for fault tolerance.

Client Library

Poseidon coordinates with DL programs via its client library. Particularly, users plug the client library into their training program, and the client library will create a syncer for each NN layer during network assembling (so that each layer one-to-one maps to one syncer), accounting for its parameter synchronization. Each syncer is then initialized, for example, setting up connections to its corresponding PS shards or (remote) peer syncers according to the coordinator’s information book, and allocating a small memory buffer for receiving remote parameter matrices or SFs, etc.

The client library manages a CPU thread pool and a GPU stream pool on the worker machine, which can be allocated by the syncer APIs when there is a syncer job created. The syncer has three main APIs, Send, Receive and Move, to be used in client programs. The Move API takes care of the memory movement between RAM and GPU memory, and performs necessary computation, e.g., the transformation between SFs and gradients, and the application of updates. It is multi-threaded using the CUDA asynchronous APIs, and will trigger an allocation from the client library’s thread/stream pools when a syncer job starts (see L14 of Algorithm 2). The Send and Receive are communication APIs that synchronize layer parameters across different model replicas. The Send API is nonblocking; it sends out parameter updates during backpropagation once they are generated, following the protocol returned by coordinator’s BestScheme API. The Receive API will be called once Send is finished. It requests either fresh parameter matrices from the KV stores or SFs from its peer syncers, and will block its current thread until it receives all of what it requested. The received messages are put into the syncer’s memory buffer for the Move API to fetch.
Managing Consistency

Poseidon focuses on synchronous parallel training which is shown to yield faster convergence compared with asynchronous training in distributed DL (as measured by wall clock time) on GPUs [11, 23]. Unless otherwise specified, our discussion in this chapter assumes synchronous replication of model parameters in each training iteration, although we note that Poseidon’s design can easily be applied to asynchronous or bounded-asynchronous consistency models [25, 60].

Poseidon implements the bulk synchronous consistency (BSP) model as follows. The client library maintains a binary vector $C$ with length the number of syncers and values reset to zeros at the start of each iteration. A syncer will set its corresponding entry in $C$ as 1 when its job finishes, and the client starts next iteration when all entries are 1. While, the KV store maintains a zero-initialized count value for each KV pair at the start of each iteration. Every time when there is an update being applied on a KV pair, its count value is increased by 1. The KV pair will be broadcasted via its Send API when its count equals to the number of workers. Poseidon handles stragglers by simply dropping them. Although asynchronous models can alleviate the straggler problem in distributed ML [60], Poseidon focuses on synchronous parallel training, because synchronous execution yields the fastest per-iteration improvement in accuracy for distributed DL (as measured by wall clock time) on GPUs [11, 23] (see §3.5.2).

3.4.2 Integrate Poseidon with DL Frameworks

For such a system to be relevant to practitioners (who may have strong preferences for particular frameworks), we would prefer not to exploit specific traits of TensorFlow’s or Caffe’s design, but should strive to be relevant to as many existing frameworks as possible.

We hence implement Poseidon to be pluggable into most existing DL frameworks to enable efficient distributed execution. Algorithm 2 provides an example. Specifically, one needs to first include Poseidon’s client library into the framework, then figure out where the backpropagation proceeds (L6), and insert Poseidon’s syner APIs in between gradient generation and application (L7). We demonstrate in §3.5.2 that with slight modifications (150 and 250 LoC for Caffe and TensorFlow), both Poseidon-enable Caffe and TensorFlow deliver linear scalings up to 32 GPU machines. Poseidon respects the programming interfaces by the native DL library and stores necessary arguments for distributed execution as environment variables to allow zero changes on the DL application programs.
Algorithm 2 Parallelize a DL library using Poseidon

1: function TRAIN(net)
2:     for iter = 1 → T do
3:         sync_count = 0
4:         net.Forward()
5:             for l = L → 1 do
6:                 net.BackwardThrough(l)
7:                     thread_pool.Schedule(sync(l))
8:             end for
9:         wait_until(sync_count == net.num_layers)
10:     end for
11: end function
12: function SYNC(l)
13:     stream = stream_pool.Allocate()
14:     syncers[l].Move(stream, GPU2CPU)
15:     syncers[l].method = coordinator.BestScheme(l)
16:     syncers[l].Send()
17:     syncers[l].Receive()
18:     syncers[l].Move(stream, CPU2GPU)
19:     sync_count++
20: end function
<table>
<thead>
<tr>
<th>Method</th>
<th>Owner</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BestScheme</td>
<td>Coordinator</td>
<td>A layer name or index</td>
<td>Get the best communication scheme of a layer</td>
</tr>
<tr>
<td>Query</td>
<td>Coordinator</td>
<td>A list of property names</td>
<td>Query information from coordinators’ information book</td>
</tr>
<tr>
<td>Send</td>
<td>Syncer</td>
<td>None</td>
<td>Send out the parameter updates of the corresponding layer</td>
</tr>
<tr>
<td>Receive</td>
<td>Syncer</td>
<td>None</td>
<td>Receive parameter updates from either parameter server or peer workers</td>
</tr>
<tr>
<td>Move</td>
<td>Syncer</td>
<td>A GPU stream and an indicator of move direction</td>
<td>Move contents between GPU and CPU, do transformations and application of updates if needed</td>
</tr>
<tr>
<td>Send</td>
<td>KV store</td>
<td>updated parameters</td>
<td>Send out the updated parameters</td>
</tr>
<tr>
<td>Receive</td>
<td>KV store</td>
<td>parameter buffer of KV stores</td>
<td>Receive gradient updates from workers</td>
</tr>
</tbody>
</table>

Table 3.5: Poseidon APIs for parameter synchronization.
3.5 Evaluation

In this section, we evaluate Poseidon’s performance on scaling up DL with distributed GPUs. We implement it into two different DL frameworks: Caffe [69] and TensorFlow [3]. We focus on the image classification task where DL is most successfully applied. Our evaluation reveals the following results:

- Poseidon has little overhead when plugged into existing frameworks; it achieves near-linear speedups across different NNs and frameworks, on up to 32 Titan X-equipped machines.
- The scheduling strategy (§3.2) substantially improves GPU and bandwidth utilization.
- The adaptive communication (§3.3) effectively alleviates the communication bottleneck, thus achieves better speedups under limited bandwidth.
- Poseidon compares favorably to other communication-reduction methods, such as the SF strategy in Adam [19], and the 1-bit quantization in CNTK [163].

3.5.1 Experiment Setup

Cluster Configuration

We conduct our experiments on a GPU cluster with each node equipped with a NVIDIA GeForce TITAN X GPU card, an Intel 16-core CPU and 64GB RAM, interconnected via a 40-Gigabit Ethernet switch. All cluster nodes have shared access to a NFS and read data through the Ethernet interface. We run our system on UBUNTU 16.04, with NVIDIA driver version 361.62, CUDA 8.0 and cuDNN v5.

Computation Engines

We deploy Poseidon on two DL frameworks, Caffe [68] and TensorFlow [3]. For Caffe, we use the official version at 2016/06/30 as the single node baseline, and modify it using Poseidon’s client library API for distributed execution. For TensorFlow, we use its open source version r0.10, and parallelize its single-node version with Poseidon’s client library, and compare to its original distributed version.

Dataset and Models

Our experiments use three well-known image classification datasets:

- CIFAR-10 [80], which contains $32 \times 32$ colored images of 10 classes, with 50K images for training and 10K for testing;

Note that as the distributed runtime of TensorFlow is highly optimized (e.g., auto-parallelization of graphs [3]) Poseidon avoids leveraging any build-in optimization of distributed TensorFlow by parallelizing its single-node version instead.
• ILSVRC12 [122], a subset of ImageNet22K that has 1.28 million of training images and 50K validation images in 1,000 categories;
• ImageNet22K [122], the largest public dataset for image classification, including 14,197,087 labeled images from 21,841 categories.

We test Poseidon’s scalability across different neural networks:
• CIFAR-10 quick: a toy CNN from Caffe that converges at 73% accuracy for classifying images in CIFAR-10 dataset;
• GoogLeNet [137]: a 22-layer CNN with 5M parameters.
• Inception-V3 [139]: the ImageNet winner, an improved version of GoogLeNet from TensorFlow;
• VGG19 [129]: A popular feature extraction network in the computer vision community [130] that has 16 CONV layers and 3 FC layers, in total 143M parameters;
• VGG19-22K: we modify the VGG19 network by replacing its 1000-way classifier with a 21841-way classifier, to classify images from the ImageNet22K dataset. The modified network has 229M parameters.
• ResNet-152 [57]: the ImageNet winner network with 152 layers.

We list their statistics and configurations in Table 3.6.

Metrics

We mainly focus on metrics that measure the system performance, such as speedups on throughput (number of images scanned per second). Our experiments focus on medium-scale distributed clusters with up to 32 machines, which distributed DL empirically benefits most from. Larger clusters require larger batch sizes, which hurt the convergence rate of each iteration [14, 23]. For completeness, we also report the statistical performance (time/epoch to converge) on ResNet-152. Poseidon uses synchronized replication which enables many models to converge in fewer steps [3, 11, 14, 23].

3.5.2 Scalability

To demonstrate Poseidon’s scalability, we train CNNs using Poseidon with different computational engines, and compare different systems in terms of their speedups on throughput. For Caffe engine, we train GoogLeNet3, VGG19, and VGG19-22K networks; for TensorFlow engine, we train Inception-V3, VGG-19, and VGG19-22K.

As there is no official implementation of Inception-V3 in Caffe, and the performance of different unofficial implementations varies dramatically, we use GoogLeNet instead.
Table 3.6: Neural networks for evaluation. Single-node batchsize is reported. The batchsize is chosen based on the standards reported in literature (usually the maximum batch size that can fill in the GPU memory is used).

<table>
<thead>
<tr>
<th>Model</th>
<th># Params</th>
<th>Dataset</th>
<th>Batchsize</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIFAR-10 quick</td>
<td>145.6K</td>
<td>CIFAR10</td>
<td>100</td>
</tr>
<tr>
<td>GoogLeNet</td>
<td>5M</td>
<td>ILSVRC12</td>
<td>128</td>
</tr>
<tr>
<td>Inception-V3</td>
<td>27M</td>
<td>ILSVRC12</td>
<td>32</td>
</tr>
<tr>
<td>VGG19</td>
<td>143M</td>
<td>ILSVRC12</td>
<td>32</td>
</tr>
<tr>
<td>VGG19-22K</td>
<td>229M</td>
<td>ImageNet22K</td>
<td>32</td>
</tr>
<tr>
<td>ResNet-152</td>
<td>60.2M</td>
<td>ILSVRC12</td>
<td>32</td>
</tr>
</tbody>
</table>

Figure 3.6: Throughput scaling when training GoogLeNet, VGG19 and VGG19-22K using Poseidon-parallelized Caffe and 40GbE bandwidth. Single-node Caffe is set as baseline (i.e., speedup = 1).
Caffe Engine

Figure 3.6 shows the throughput vs. number of workers when training the three networks using Caffe engine, given 40GbE Ethernet bandwidth available. We compare the following systems:

- **Caffe**: unmodified Caffe that executes on a single GPU;
- **Caffe+PS**: we parallelize Caffe using a vanilla PS, i.e., the parameter synchronization happens sequentially after the backpropagation in each iteration;
- **Caffe+WFBP**: parallelized Caffe using Poseidon so the communication and computation are overlapped. However, we disable the adaptive communication so that parameters are synchronized only via PS;
- **Poseidon**: the full version of Poseidon-Caffe.

Poseidon shows little overheads when combined with Caffe; running on a single node with no communication involved, Poseidon-Caffe can process 257, 35.5 and 34.2 images per second when training GoogLeNet, VGG19 and VGG19-22K, respectively, as compared to the original Caffe, which can process 257, 35.5 and 34.6 images, and Caffe+PS, which can only process 213.3, 21.3 and 18.5 images per second, due to the overheads caused by memory copy operations between RAM and GPU, which have been overlapped by Poseidon with the computation. In distributed environment, the rescheduling of computation and communication significantly improves the throughput: when training GoogLeNet and VGG19, incorporating WFBP achieves almost linear scalings up to 32 machines, and for the larger VGG19-22K network, Caffe+WFBP achieves 21.5x speedup on 32 machines. We conclude that rescheduling and multi-threading the communication and computation are key to the performance of distributed DL on GPUs, even when the bandwidth resource is abundant. Poseidon provides an effective implementation to overlap these operations for DL frameworks, to guarantee better GPU utilization.

When the available bandwidth is sufficient, the adaptive communication strategy shows small improvement on training GoogLeNet and VGG19. However, when training VGG19-22K which has three FC layers that occupy 91% of model parameters, it improves over Caffe-WFBP from 21.5x to 29.5x on 32 nodes.

TensorFlow Engine

We also modify TensorFlow using Poseidon, and compare the following systems in terms of speedup on throughput:

- **TF**: TensorFlow with its original distributed execution runtime;
- **TF+WFBP**: we modify TensorFlow using Poseidon’s client library. Specifically, we change the *assign* operator in TensorFlow, so that instead of being applied, the parameter updates will be synchronized via *Poseidon’s PS interface with WFBP*;
- **Poseidon**: the full version of Poseidon-parallelized TensorFlow with the adaptive communication enabled.

28
Figure 3.7: Throughput scaling when training Inception-V3, VGG19 and VGG19-22K using Poseidon-parallelized TensorFlow and 40GbE bandwidth. Single-node TensorFlow is set as baseline (i.e., speedup = 1).

We train Inception-V3, VGG19 and VGG19-22K models and report the results in Figure 3.7. Running on a single node, Poseidon processes 43.2, 38.2 and 34.5 images per second on training Inception-V3, VGG19 and VGG19-22K, while original TensorFlow processes 43.2, 38.5 and 34.8 images per second on these three models, respectively – little overhead is introduced by our modification. In distributed execution, Poseidon achieves almost linear speedup on up to 32 machines. Distributed TensorFlow, however, demonstrates only 10x speedup on training Inception-V3 and even fails to scale on training the other two networks in our experiments. To investigate the problem of TensorFlow and explain how Poseidon improves upon it, we illustrates in Figure 3.8 the (averaged) ratio of busy and stall time of a GPU when training the three networks using different systems on 8 nodes. Observe that Poseidon keeps GPUs busy in most of the time, while TensorFlow wastes much time on waiting for parameter synchronization. The inefficiency of distributed TensorFlow stems from two sources. First, TensorFlow partitions model parameters in a coarse-grained granularity – each tensor (instead of a KV pair) in the model is
assigned to a PS shard. A big tensor (such as the parameter matrix in VGG19) is highly likely to create communication bottleneck on its located server node. Poseidon fixes this problem by partitioning parameters among server nodes in a finer-grained granularity using KV pairs, so that every node has evenly distributed communication load; as an evidence, TF+WFBP demonstrates higher computation-to-stall ratio in Figure 3.8. Second, TensorFlow cannot reduce the communication overheads while the suggested adaptive communication approach effectively reduces the size of messages. As a result, Poseidon further improves upon TF+WFBP from 22x to 30x on 32 nodes.

Multi-GPU Settings

The scheduling and communication strategies can be directly extended to support distributed multi-GPU environment with minor modifications. Specifically, when there are more than 1 GPU on a worker node, Poseidon will first collect the gradient updates following WFBP locally (either by full matrices or SFs) from multiple GPUs to a leader GPU using CudaMemcpy() API. If those updates are determined to be communicated via full matrices, Poseidon will aggregate them locally before sending out. Using Caffe engine on a single node, Poseidon achieves linear scalings on up to 4 Titan X GPUs when training all three networks, outperforming Caffe’s multi-GPU version, which shows only 3x and 2x speedups when training GoogLeNet and VGG19. When running on AWS p2.8xlarge instances (8 GPUs each node), Poseidon reports 32x and 28x speedups when training GoogLeNet and VGG19 with 4 nodes (32 GPUs in total), confirming the existence of the overheads caused by memory movement between GPUs, though less substantial than network communication\(^4\). We will tackle this issue in Chapter 4.

Statistical Performance

For completeness, we report in Figure 3.9 the statistical performance for training ResNet-152 using Poseidon. Poseidon achieves near-linear speedups on both system throughput and statistical convergence: Poseidon delivers 31x speedup in terms of throughput, and reaches 0.24

\(^4\)Note that the K80 GPUs on p2.8xlarge has less GFLOPS than Titan X used in our main experiments – the communication burden is less severe.
reported error with less than 90 epochs with both 16 and 32 nodes – thus linear scales in terms of time to accuracy, compared to 8 nodes with batchsize = $32 \times 8$, which is a standard setting as in [57], echoing recent results that synchronous training on distributed GPUs yields better performance than asynchronous training in terms of time to quality for most NNs [11, 23]. For other NNs in Table 3.6, Poseidon delivers the same quality of accuracy as reported in their papers [82, 130, 137, 139] on up to 32 GPUs.

### 3.5.3 Bandwidth Experiments

To understand the effectiveness of the proposed adaptive communication strategy, we create an ablation environment where network bandwidth is limited. We use Linux traffic control tool tc to lower the available bandwidth on each node, and compare the training throughput between with and without the adaptive communication. We focus on Caffe engine in this section because it is
Figure 3.11: Throughput scaling when training GoogLeNet, VGG19 and VGG19-22K using Poseidon-parallelized Caffe with varying network bandwidth. Single-node Caffe is set as baseline (speedup = 1).

Figure 3.11 plots the speedup on throughput vs. number of workers when training GoogLeNet, VGG19 and VGG19-22K with different maximum bandwidth. Clearly, limited bandwidth prevents a standard PS-based system from linearly scaling with the number of nodes; for example, given 10GbE bandwidth (which is a commonly-deployed Ethernet configuration in most cloud computing platforms), training VGG19 using PS on 16 nodes can only be accelerated by 8x. This observation confirms our argument that limited bandwidth would result in communication bottleneck when training big models on distributed GPUs. Fortunately, Poseidon significantly alleviates this issue. Under limited bandwidth, it constantly improves the throughput by directly reducing the size of messages needed to be communicated, especially when the batch size is small; when training VGG19 and VGG19-22K, Poseidon achieves near-linear speedup on 16 machines using only 10GbE bandwidth, while an optimized PS would otherwise need 30GbE or even higher to achieve. Note that Poseidon will never underperform a traditional PS scheme be-
cause it will reduce to a parameter server whenever it results in less communication overheads; for instance, we observe that Poseidon reduces to PS when training GoogLeNet on 16 nodes, because GoogleNet only has one thin FC layer (1000 × 1024) and is trained with a large batch size (128).

### 3.5.4 Comparison to Other Methods

In this section, we compare Poseidon against other communication reduction methods, including Adam [19] and CNTK 1-bit quantization [163], and discuss their pros and cons.

**Comparison to Adam [19]**

To save bandwidth, Adam [19] synchronizes the parameters of a FC layer by first pushing SFs generated on all workers to a PS node, and then pulling back the full parameter matrices thereafter. As direct comparisons to Adam [19] are inaccessible, we implement its strategy in Poseidon, and compare it (denoted as Adam) to TF-WFBP and Poseidon by monitoring the network traffic of each machine when training VGG19 on 8 nodes using TensorFlow engine. As shown in Figure 3.10, the communication workload is highly imbalanced using Adam’s strategy. Unlike a traditional PS (TF-WFBP) where the parameters are equally distributed over multiple shards, Adam cannot partition the parameters of FC layers because of their usage of SFs. Although the push operation uses SFs to reduce message size, the pull requires some server nodes to broadcast big matrices to each worker node, which creates bursty traffic that results in communication bottleneck on them. By contrast, Poseidon either partitions parameters equally over multiple PS shards, or transmits SFs among all peer workers, both are communication load-balanced and avoid bursty communication situations. Quantitatively, Adam delivers 5x speedup with 8 nodes when training VGG19.

**Comparison to CNTK [163]**

We compare Poseidon to the 1-bit quantization technique proposed in CNTK [163]. We create a baseline Poseidon-1bit which uses the 1-bit strategy to quantize the gradients in FC layers, and add the residual to updates of the next iteration. We then train the CIFAR-10 quick network, and plot the training loss and test error vs. iterations for two systems (both have linear scaling on throughput). As in Figure 3.12, 1-bit quantization yields worse convergence in terms of accuracy – on 4 GPUs, it achieves 0.5 error after 3K iterations, while Poseidon quickly converges to 0.3 error at iteration 1000. We conjecture this is caused by the quantization residual, which seems to be equivalent to delayed updates that may hurt the convergence performance when training NNs on images, confirmed by Cui et al. [23]. We also directly train VGG19 using CNTK-1bit system, and report 5.8x, 11x, 20x speedups on 8, 16 and 32 nodes, respectively, thus less scale-ups than Poseidon, and also compromised statistical performance due to approximate updates.
Figure 3.12: Training loss and test error vs. iteration when training CIFAR-10 quick network using *Poseidon* and *Poseidon-1bit* on 4 GPUs with Caffe engine.

<table>
<thead>
<tr>
<th>Framework</th>
<th>Data</th>
<th># machines/cores</th>
<th>Time</th>
<th>Train acc</th>
<th>Test acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poseidon</td>
<td>7.1M ImageNet22K for training, 7.1M for test</td>
<td>8 / 8 GPUs</td>
<td>3 days</td>
<td>41%</td>
<td>23.7%</td>
</tr>
<tr>
<td>Adam [19]</td>
<td>7.1M ImageNet22K for training, 7.1M for test</td>
<td>62 machines/?</td>
<td>10 days</td>
<td>N/A</td>
<td>29.8%</td>
</tr>
<tr>
<td>MxNet [14]</td>
<td>All ImageNet22K images for training, no test</td>
<td>1/4 GPUs</td>
<td>8.5 days</td>
<td>37.19%</td>
<td>N/A</td>
</tr>
<tr>
<td>Le et al. [85] w/pretrain</td>
<td>7.1M ImageNet 22K, 10M unlabeled images for training, 7.1M for test</td>
<td>1K/16K CPU cores</td>
<td>3 days</td>
<td>N/A</td>
<td>15.8%</td>
</tr>
</tbody>
</table>

Table 3.7: Comparisons of the image classification results on ImageNet 22K.

### 3.5.5 Application: Scaling Up Image Classification on ImageNet 22K

ImageNet 22K was the largest public dataset for image classification (2017), including nearly 14.2M labeled images from 21,841 categories, which was rarely touched by the research community due to its massive data size and complexity. We experiment on ImageNet 22K to demonstrate the scalability of Poseidon. As no official test data exists for evaluation, following previous settings in [19, 28, 85], we randomly split the whole set into two parts, and use the first 7.1 million of images for training and remained for test. Similar to ILSVRC 2012, we resize all images to $256 \times 256$ and report the top-1 test accuracy.

**Settings**

We design an AlexNet-like architecture; specifically, the CNN takes a random $227 \times 227$ crop from the original image, and forwards it into 5 CONV layers and 2 FC layers before prediction.
The CNN has convolution filters with sizes $7 \times 7$, $5 \times 5$, and $3 \times 3$. Similar to AlexNet, the first, second and fifth CONV layers are followed by max pooling layers with size $3 \times 3$ and stride 2. Two FC layers with 3,000 neurons each are put at the top of the network, followed by a softmax layer to be a 21,841-way classifier with 120M parameters and 1.8 billion of connections overall. We train the CNN with data-parallelism by equally partitioning the training data into 8 GPU nodes. The batch size and staleness are fixed at 256 and 0, respectively. The network is trained using the step learning rate policy, with base learning rate 0.005 and decreased 6 times.

**Performance**

Table 3.7 compares our result to those of previous work on ImageNet 22K: Adam [19], MxNet [14], and Le et al. [85]. Note that at this point fair comparisons between different frameworks might not be possible, because the experiment setup of ImageNet 22K was not standardized, all the source codes were not fully available yet, and large variations might exist in system configurations, models, and implementation details. However, it is clear that Poseidon achieves a competitive accuracy 23.7% with the state-of-the-art systems with shorter training time and less machine resources. Compared to Adam [19], we only use 30% training time and 13% machines to achieve 23.7% accuracy with a similarly sized model. Promisingly, we achieve a higher training accuracy with 3 days of training using a well-established CNN model — this compares favorably to MXNet, which uses the whole set of 14.1 million images to train an inception-BN structure [66] using 4 GPUs in a single machine without network communication, and reports 37.1% train accuracy after 8.5 days of training.

### 3.6 Additional Related Work

**PS-based Distributed DL Systems**

Based on the parameter server [87, 152] architecture, a number of CPU-based distributed DL systems have been developed, such as [28, 85, 150, 175] and Adam [19]. They are purely PS-based systems on CPU-only clusters, whereas Poseidon addresses the more challenging case of GPU clusters.

Scaling up DL on distributed GPUs is an active field of research. Coates et al. [21] build a GPU-based multi-machine system for DL using model parallelism rather than data parallelism, and their implementation is rather specialized for a fixed model structure while demanding specialized hardware, such as InfiBand networking. TensorFlow [3] is Google’s distributed ML platform that uses a dataflow graph to represent DL models, and synchronizes model parameters via PS. It therefore cannot dynamically adjust its communication method depending on the layer and cluster information as Poseidon does. MXNet [14] is another DL system that uses PS for distributed execution, and supports TensorFlow-like graph representations for DL models. By auto-parallelizing independent subgraphs, both frameworks implicitly overlap the communication and computation. By contrast, Poseidon has a more explicit way to overlap them via its client library. Hence, Poseidon can be also used to parallelize non-graph-based frameworks. Moreover, both MXNet and TensorFlow do not address the bottleneck caused by limited network bandwidth, which undermines their scalability when training large models with dense
layers (e.g., big softmax). Besides, Cui et al. propose GeePS [23] that manages the limited GPU memory and report speedups on distributed GPUs. While, GeePS does not address the issue of limited network bandwidth. Therefore, Poseidon’s technique could be combined with them to enable better training speedups. Also of note are several efforts to port Caffe onto other distributed platforms, such as SparkNet [103], YahooCaffe [160] and FireCaffe [64], the former reports a 4-5 times speedup with 10 machines (and hence less scalability than our results herein).

Other Distributed ML Systems

CNTK [163] is a DL framework that supports distributed executions and addresses the problem of communication bottleneck via the 1-bit quantization technique. CNTK demonstrates little negative impact on convergence in speech domains [124, 125]. However, in may scenarios (§3.5.4), the performance is usually compromised by noisy gradients [3, 23]. By contrast, Poseidon’s HybComm reduces the communication while always guaranteeing synchronous training. There are also growing interest in parallelizing ML applications using peer-to-peer communication, such as MALT [86], SFB [155] and Ako [151]. Poseidon draws inspiration from these works but goes one step further as it is an adaptive best-of-both-worlds protocol, which will select client-server communication whenever it would result in fewer overheads.

Recently, there is a surge of interest in bring in collective communication primitives, such as AllReduce, AllGather, from the high-performance computing areas into distributed deep learning [46, 67, 126]. Specialized systems built on top of these libraries (e.g. MPI) have successfully scaled ResNet training on thousands of GPU nodes on a homogeneous cluster, and reduced the ImageNet training into hours or even minutes. The proposed techniques in scheduling and communication can be integrated with collective communication, similarly as we have done with parameter server and sufficient factor broadcasting architectures.

Adaptive Communication

A second successful instance of adaptive communication [76] is to mix parameter server with AllReduce [76], based on the sparsity of trainable variables in NNs – defined depending on how their elements are accessed in computation. For a dense variable, all elements are accessed at least once during a single training iteration, which is the case for most build blocks in CNNs. For a sparse variable, only a subset of the elements are accessed in one iteration – a pattern that is commonly found in NLP models with embedding layers, where the embedding variable could be as large as with billions of float parameters. Synchronizing such a variable across multiple GPUs requires significant network bandwidth and consumes many CPU clocks for aggregating results from GPUs, and if done in the same way as for dense variables, usually results in no scalability. While on the other hand, treating all variables as sparse variables is sub-optimal, as there are highly optimized implementations for communicating dense variables across GPUs such as the NCCL library. The idea therefore comes in naturally – estimates the communication overhead ahead of execution, and adaptively chooses from PS and AllReduce based on the spare access patterns of different layers and estimated overheads.
Scheduling for Distributed DL

WFBP, as one of the earliest structure-aware scheduling for distributed DL, has inspired a few follow-up work, such as ByteScheduler [115] and PipeDream [55]. ByteScheduler analyzes and derives the optimal schedule in the ideal scenario with strong assumptions, and uses Bayesian Optimization (BO) to auto-tune the partition size to work with a practical environment. PipeDream proposes Pipeline parallelism that automatically partitions DNN training across workers, combining inter-batch pipelining with intra-batch parallelism to better overlap computation with communication, while minimizing the amount of data communicated.
Chapter 4

Memory Management

The limited size of GPU device memory was viewed as a serious impediment to parallel DL training on GPU clusters, limiting the size of the model to what could fit in a single device memory. In fact, as observed by prior work [19, 21, 81], this would seem to imply that GPU-based systems (with their limited GPU memory) are suited only for relatively small neural networks. Besides, the frequent data movements between GPU and DRAM also adds nontrivial overheads to both communication and computation.

In this chapter, we derive strategies and system implementations to improve memory management in distributed ML. We first present memory swapping (MemSwap), a simple memory management mechanism inspired by a key observation in DL training – a layer's computation only depends on the outputs (and intermediate states) produced by its dependent layers, but not that by the entire model. Informed by the dependency structure presented in the model, the memory management can be adapted to hide data movement overheads, and made it possible for training large models that otherwise cannot fit in the limited GPU memory.

We then present a corresponding implementation, GeePS [23], a parameter server system specialized for scaling deep learning applications across GPUs distributed among multiple server machines. Based on MemSwap, GeePS performs a number of optimizations specially tailored to making efficient use of GPUs, such as GPU-friendly caching, data staging, and memory management techniques – GeePS overcomes the above-mentioned limitations by assuming control over memory management and placement, and carefully orchestrating data movement between CPU and GPU memory based on its observation of the access patterns at each layer of the neural network.

Combining the techniques presented in Chapter 3 and in this chapter enables designing and employing CNN models with a much greater number of parameters (which otherwise might be difficult due to the significantly increased communication loads and memory footprints). In §?? of this chapter, we briefly introduce a new CNN architecture, hierarchical deep convolutional neural networks (HD-CNN), built on top of GeePS and Posedion. By augmenting a backbone model with 2x more auxiliary parameters and 7x more intermediate states, HD-CNN boosts the image classification accuracy on ImageNet for up to 3 percentage, and had achieved state-of-the-art results in 2016.
4.1 Introduction

This section walks through deep learning on GPU and GPU clusters from the perspective of memory usage and consumption, and then raises the specific challenges in the aspect of memory management.

4.1.1 Deep Learning on GPUs: a Memory Management Perspective

GPUs are often used to train deep neural networks, because the primary computational steps match their SIMD-style nature and they provide much more raw computing capability than traditional CPU cores. Most high end GPUs are on self-contained devices that can be inserted into a server machine, as illustrated in Figure 4.1. One key aspect of GPU devices is that they have dedicated local memory, which we will refer to as GPU memory, and their computing elements are only efficient when working on data in that GPU memory. Data stored outside the device, in CPU memory, must first be brought into the GPU memory (e.g., via PCI DMA) for it to be accessed efficiently.

![Diagram of a machine with a GPU device.](image)

Neural network training is an excellent match to the GPU computing model. NVIDIA provides a cuBLAS library [1], which can be used to launch basic linear algebra GPU computations from CPU code, and a cuDNN library [2] for launching GPU computations more specific to neural networks, such as calculating convolutions.

Contemporary ML frameworks such as Caffe, TensorFlow, PyTorch, MxNet all follow a similar pattern to interact with GPU devices and their memory: a single-threaded worker launches and joins with GPU computations, by calling NVIDIA cuBLAS and cuDNN libraries as well as some customized CUDA kernels. Each mini-batch of training data is read from an input file via the CPU, moved to GPU memory, and then processed as described above. For efficiency, many
frameworks keep all model parameters and intermediate states in the GPU memory. As such, it is effective only for models and mini-batches small enough to be fully held in GPU memory. Figure 4.2 illustrates the CPU and GPU memory usage adopted in these frameworks.

Figure 4.2: Single GPU deep learning and memory usage on contemporary ML frameworks.

4.1.2 Memory Challenges for Distributed DL on GPU Clusters

Parameter server architecture has become a popular approach to making it easier to build and scale DL across CPU-based clusters [4, 19, 22, 28, 60, 116], particularly for data-parallel execution. Figure 4.3 illustrates the basic parameter server architecture from a memory perspective – all states shared among worker threads (i.e., the model parameters being learned) are kept in distributed shared memory implemented as a specialized key-value store called a “parameter server”, usually on DRAM. While the picture illustrates the parameter server as separate from the machines executing worker threads, and some systems do work that way, the server-side parameter server state is commonly sharded across the same machines as the worker threads. The latter approach is particularly appropriate when considering a parameter server architecture for GPU-based ML execution, since the CPU cores and CPU memory would be largely unused by the GPU-based workers.

Given its proven value in CPU-based distributed ML, it is natural to use the same basic architecture and programming model with distributed ML on GPUs. To explore its effectiveness, we ported two applications (based on Caffe) to a state-of-the-art CPU-based parameter server system (IterStore [22]). Doing so was straightforward and immediately enabled distributed deep learning on GPUs, confirming the application programmability benefits of the data-parallel pa-
Parameter cache
...Worker Worker
Machine
...
Parameter Server
Parameter data

Figure 4.3: Parallel ML with parameter server. Parameter cache or data usually locate on shared CPU memory (DRAM).

Parameter server approach. Figure 4.4 illustrates what sits where in memory, to allow comparison to Figure 4.2 and designs described later.

While it was easy to get working, the performance was not acceptable. As noted by Chilimbi et al. [19], the GPU’s computing structure makes it “extremely difficult to support data parallelism via a parameter server” using current implementations, because of GPU stalls, insufficient synchronization/consistency, or both. Also as noted by them and others [154], the need to fit the full model, as well as a mini-batch of input data and intermediate neural network states, in the GPU memory limits the size of models that can be trained. While Chapter 3 has proposes solutions to address issues related with communication and scheduling, the next section describes design for overcoming the limitations in memory.

4.2 Memory Swapping

The section describe memory swapping (MemSwap), and a series of techniques developed based on it to enable distributed training of large models on limited GPU device memory. The key idea of memory swapping is simple – when the GPU memory of a machine is not big enough to host all data, including parameters and intermediate states, we swap part of the data to the CPU memory. For efficiency, we restrict the ML training to still access everything through GPU memory, as before, and the memory management library will do the data movement between GPU and CPU when needed. Copying data between GPU and CPU memory could significantly
Figure 4.4: Distributed ML on GPUs using a CPU-based parameter server. The right side of the picture is much like the single-GPU illustration in Figure 4.2. But, a parameter server shard and client-side parameter cache are added to the CPU memory, and the parameter data originally only in the GPU memory is replaced in GPU memory by a local working copy of the parameter data. Parameter updates must be moved between CPU memory and GPU memory, in both directions, which requires an additional application-level staging area since the CPU-based parameter server is unaware of the separate memories.
slow down data access. To minimize slowdowns, a memory manager will use separate threads to perform the memcpy operations between CPU and GPU memory in the background. In order to prefetch the content from CPU to GPU, it will need to know in advance the sets of parameter data that the application will access. This could be enabled by incorporating knowledge of the neural network structure into memory management. For example, in the forward pass of CNNs, once the forward computation $f_i$ (defined in §3 and Figure 3.3) is finished, the intermediate states or parameters stored before layer $i$ could be temporally moved to CPU memory as they are no longer needed until the backward pass comes back at layer $i$; and this movement could be performed concurrently with the ongoing forward computation $f_j (j > i)$, upcoming backward computation $b_j (j > i)$, and communication $o_j, i_j (j > i)$. Similarly, before the backward computation approaches layer $i$, the data movement for the intermediate states needed for $b_i$, from CPU memory back to GPU memory, can be performed in advance and overlapped with the backward computation operations $b_j (j > i)$.

We next describe an implementation of MemSwap on parameter server to enable efficient support of parallel ML applications running on distributed GPUs, but note MemSwap, as a generic memory management strategy, can be used in any non-distributed setting or Non-CPU devices with limited memory. In fact, since the proposal of MemSwap and GeePS, we have observed extensive development on using MemSwap to improve the memory management in deep learning [62].

We describe three primary specializations to a parameter server to enable efficient support of parallel ML applications running on distributed GPUs: explicit use of GPU memory for the parameter cache, batch-based parameter access methods, and parameter server management of GPU memory on behalf of the application. The first two address performance, and the third expands the range of problem sizes that can be addressed with data-parallel execution on GPUs.

### 4.2.1 Maintaining the Parameter Cache in GPU memory

One important change needed to improve parameter server performance for GPUs is to keep the parameter cache in GPU memory, as shown in Figure 4.5 (Section 4.2.3 discusses the case where everything does not fit). Perhaps counter-intuitively, this change is not about reducing data movement between CPU memory and GPU memory — the updates from the local GPU must still be moved to CPU memory to be sent to other machines, and the updates from other machines must still be moved from CPU memory to GPU memory. Rather, moving the parameter cache into GPU memory enables the parameter server client library to perform these data movement steps in the background, overlapping them with GPU computing activity. Then, when the application uses the read or update functions, they proceed within the GPU memory. Putting the parameter cache in GPU memory also enables updating of the parameter cache state using GPU parallelism.

### 4.2.2 Pre-built Indexes and Batch Operations

Given the SIMD-style parallelism of GPU devices, per-value read and update operations of arbitrary model parameter values can significantly slow execution. In particular, performance problems arise from per-value locking, index lookups, and data movement. To realize sufficient performance, GPU-specialized parameter server needs to support batch-based interfaces for reads
Figure 4.5: Parameter cache in GPU memory. In addition to the movement of the parameter cache box from CPU memory to GPU memory, this illustration differs from Figure 4.4 in that the associated staging memory is now inside the parameter server library. It is used for staging updates between the network and the parameter cache, rather than between the parameter cache and the GPU portion of the application.

(e.g., READ–BATCH) and updates (e.g., UPDATE–BATCH). Moreover, we exploit the repeating nature of iterative model training [22] to provide batch-wide optimizations, such as pre-built indexes for an entire batch that enable GPU-efficient parallel “gathering” and updating of the set of parameters accessed in a batch. These changes make parameter servers much more efficient for GPU-based training.

4.2.3 Managing Limited GPU Device Memory

As noted earlier, the core idea of MemSwap is to manage the GPU memory for the application and swap the data that is not currently being used to CPU memory. It can move the data between GPU and CPU memory in the background, minimizing overhead by overlapping the transfers with the training computation, and our results demonstrate that the two do not interfere with one another.

Managing GPU Memory Inside the Parameter Server

Incorporating MemSwap requires parameter server to provide read and update interfaces with parameter-server-managed buffers. When the application reads parameter data, the parameter server client library will allocate a buffer in GPU memory for it and return the pointer to this
buffer to the application, instead of copying the parameter data to a buffer provided by the application. When the application finishes using the parameter data, it returns the buffer to the parameter server. We call those two interfaces \texttt{BUFFER-READ-BATCH} and \texttt{POST-BUFFER-READ-BATCH}. When the application wants to update parameter data, it will first request a buffer from the parameter server using \texttt{PRE-BUFFER-UPDATE-BATCH} and use this buffer to store its updates. The application calls \texttt{BUFFER-UPDATE-BATCH} to pass that buffer back, and the parameter server library will apply the updates stored in the buffer and reclaim the buffer memory. To make it concise, in the rest of this chapter, we will refer to the batched interfaces using PS-managed buffers as \texttt{READ}, \texttt{POST-READ}, \texttt{PRE-UPDATE}, and \texttt{UPDATE}.

The application can also store their local non-parameter data (e.g., intermediate states) in the parameter server using the same interfaces, but with a \texttt{LOCAL} flag. The local data will not be shared with the other application workers, so accessing the local data will be much faster than accessing the parameter data. For example, when the application reads the local data, the parameter server will just return a pointer that points to the stored local data, without copying it to a separate buffer. Similarly, the application can directly modify the requested local data, without needing to issue an \texttt{UPDATE} operation.

**Swapping Data to CPU Memory When It Does Not Fit**

By storing the local data in the parameter server, almost all GPU memory can be managed by the parameter server client library. When the GPU memory of a machine is not big enough to host all data, the parameter server will store part of the data in the CPU memory. The application still accesses everything through GPU memory, as before, and the parameter server library will do the data movement for it. When the application \texttt{READs} parameter data that is stored in CPU memory, the parameter server will perform this read using a CPU core and copy the data from CPU memory to an allocated GPU buffer. Likewise, when the application \texttt{READs} local data that is stored in CPU memory, the parameter server will copy the local data from CPU memory to an allocated GPU buffer. Figure 4.6 illustrates the resulting data layout in the GPU and CPU memories.

**GPU/CPU Data Movement in the Background**

Copying data between GPU and CPU memory could significantly slow down data access. To minimize slowdowns, MemSwap uses separate threads to perform the \texttt{READ} and \texttt{UPDATE} operations in the background. For an \texttt{UPDATE} operation, because the parameter server owns the update buffer, it can apply the updates in the background and reclaim the update buffer after it finishes. In order to perform the \texttt{READ} operations in the background, the parameter server will need to know in advance the sets of parameter data that the application will access. Fortunately, as we have explained above, this could be inferred based on the knowledge of the model structure, so the parameter server can easily predict the \texttt{READ} operations and perform them in advance in the background.
Figure 4.6: Parameter cache and local data partitioned across CPU and GPU memories. When all parameter and local data (input data and intermediate states) cannot fit within GPU memory, our parameter server can use CPU memory to hold the excess. Whatever amount fits can be pinned in GPU memory, while the remainder is transferred to and from buffers that the application can use, as needed.

4.3 GeePS: Memory-optimized Parameter Server on GPUs

This section describes GeePS, a GPU-specialized parameter server system that optimizes memory management based on MemSwap described in Section 4.2.

4.3.1 GeePS Data Model and API

GeePS is a C++ library that manages both the parameter data and local data for machine learning applications. The distributed application program usually creates one ML worker process on each machine and each of them links to one instance of the GeePS library. For GPU-based ML applications (such as Caffe or TensorFlow), the worker often runs in a single CPU thread and launches and joins with GPU computations, which might be NVIDIA library calls or customized CUDA kernels. On initializing the GeePS library, the application will provide the list of hosts that the program is running on, and each GeePS instance will create connections to the other hosts.

GeePS manages data as a collection of rows indexed by keys. The rows are then logically grouped into tables, and rows in the same table share the same attributes (e.g., data age). GeePS implements the read and update operations with parameter-server-managed buffers for parameter data access, and we call them READ, POST-READ, PRE-UPDATE, and UPDATE for short. For
local data access, GeePS provides **LOCAL-ACCESS** and **POST-LOCAL-ACCESS**, for which the application can directly modify the accessed local data without an explicit update operation. GeePS also provides a **TABLE-CLOCK** function for application workers to signal the completion of per-table updates. GeePS implicitly synchronizes the application workers by letting them wait on read operations, when the data is not available yet. It supports three execution synchrony models: BSP, SSP [60], and Asynchrony.

### 4.3.2 Architecture

#### Storing Data

Each GeePS instance stores one shard of the master version of the parameter data in its **parameter server shard**. The parameter server shard is not replicated, and fault tolerance is handled by checkpointing. In order to reduce communication traffic, each instance has a **parameter cache** that stores a local snapshot of the parameter data, and the parameter cache is refreshed from the parameter server shards, such as at every clock for BSP. When the application applies updates to the parameter data, those updates are also stored in the parameter cache (a write-back cache) and will be submitted to the parameter server shards at the end of every clock. The parameter cache has two parts, a GPU-pinned parameter cache and a CPU parameter cache. If everything fits in GPU memory, only the GPU parameter cache is used. But, if the GPU memory is not big enough, GeePS will keep some parameter data in the CPU parameter cache. The data placement policies are described in Section 4.3.4. Each GeePS instance also has an **access buffer pool** in GPU memory, and GeePS allocates GPU buffers for **READ** and **PRE-UPDATE** operations from the buffer pool. When **POST-READ** or **UPDATE** operations are called, the memory will be reclaimed by the buffer pool. GeePS manages application’s input data and intermediate states as **local data**, which is local to each worker. The local data also has a GPU-pinned part and a CPU part, with the CPU part only used if necessary. GeePS divides the key space into multiple **partitions**, and the rows in different partitions are managed in separate data structures, with different sets of communication threads.

#### Data Movement across Machines

GeePS performs communication across machines asynchronously with three types of communication threads: **keeper threads** manage the parameter data in parameter server shards; **pusher threads** move parameter data from parameter caches to parameter server shards, by sending messages to keeper threads; **puller threads** move parameter data from parameter server shards to parameter caches, by receiving messages from keeper threads.

The communication is implemented using sockets, so the data needs to be copied to some CPU staging memory before being sent through the network, and the received data will also be in the CPU staging memory. To send/receive data from/to GPU parameter cache, the pusher/puller threads will need to move the data between CPU memory and GPU memory using CUDA APIs.
Data Movement inside a Machine

GeePS uses two background threads to perform the READ and UPDATE operations for the application. The allocator thread performs the READ and PRE-UPDATE operations by allocating buffers from the buffer pool and (only for READ) copying parameter values from the parameter cache to the buffers. The reclaimer thread performs the POST-READ and UPDATE operations by (only for UPDATE) applying updates from the buffers to the parameter cache and reclaiming the buffers back to the buffer pool. These threads assign and update parameter data in large batches with pre-built indices by launching CUDA kernels on GPUs, as described in Section 4.3.3.

Locking

GeePS’s background threads synchronize with each other, as well as the application threads, using mutex locks and conditional variables. Unlike some other CPU-based parameter servers that use per-row locks [22, 152], we use a coarse-grained locking design, where one set of mutex lock and conditional variable are used for a whole key partition. We make this design decision for two reasons. First, with coarse-grained locking, batched data operations can be easily performed on a whole partition of rows. Second, unlike CPU applications, where one application thread is launched for each CPU core, a GPU application often has just one CPU host thread accessing the parameter server, making lock contention less of an issue.

Operation Sequence Gathering

Some of the specializations (pre-built indices, background READ, and data placement decisions) exploit knowledge of the operation sequence of the application. GeePS implements an operation sequence gathering mechanism like that described by Cui et al. [22]. It can gather the operation sequence information either by directly analyzing the model structure, or in the first iteration or in a virtual iteration, during which the application just reports its sequence of operations without doing any real computation or keeping any states. GeePS uses the gathered operation sequence information as a hint to build the data structures (parameter server shard, parameter cache, and local data), build the access indices, prefetch the data (including cross-machine data fetching and background READ), and make GPU/CPU data placement decisions. Since the gathered access information is used only as a hint, knowing the exact operation sequence is not a requirement for correctness, but a performance optimization.

For most deep learning applications (including CNN and RNN), the application accesses all model parameters every mini-batch. For some applications with sparse training data (e.g., BOW representation for NLP tasks), the bottom layer of the network might just use a subset of the weights. Even for these tasks, the operation sequence of a whole epoch still repeats. The operation sequence only changes when the training data is shuffled. For this special case, we just prefetch all the parameter data.

4.3.3 Parallelizing Batched Access

GeePS provides a key-value store interface to the application, where each parameter row is named by a unique key. When the application issues a batched read or update operation, it
will provide a list of keys for the target rows. GeePS could use a hash map to map the row keys to the locations where the rows are stored. But, in order to make the batched access be executed by all GPU cores, GeePS will use the following mechanism. Suppose the application update \( n \) rows, each with \( m \) floating point values, in one UPDATE operation, it will provide an array of \( n \) keys \( \{ keys[i] \}_{i=1}^{n} \) and an array of \( n \) parameter row updates \( \{ \{ updates[i][j] \}_{j=1}^{m} \}_{i=1}^{n} \). GeePS will use an index with \( n \) entries, where \( \{ index[i] \}_{i=1}^{n} \) is the location of the parameter data for \( \{ keys[i] \}_{i=1}^{n} \). Then, it will do the following data operation for this UPDATE: \( \{ \{ parameters[index[i]][j] += updates[i][j] \}_{j=1}^{m} \}_{i=1}^{n} \). This operation can be executed with all the GPU cores. Moreover, the index can be built just once for each batch of keys, based on the operation sequence gathered as described above, and re-used for each instance of the given batch access.

### 4.3.4 GPU Memory Management

GeePS keeps the GPU-pinned parameter cache, GPU-pinned local data, and access buffer pool in GPU memory. They will be all the GPU memory allocated in a machine if the application stores all its input data and intermediate states in GeePS and uses the GeePS-managed buffers. GeePS will pin as much parameter data and local data in GPU memory as possible. But, if the GPU memory is not large enough, GeePS will keep some of the data in CPU memory (the CPU part of the parameter cache and/or CPU part of the local data).

In the extreme case, GeePS can keep all parameter data and local data in the CPU memory. But, it will still need the buffer pool to be in the GPU memory, and the buffer pool needs to be large enough to store all the actively used data even at peak usage. We refer to this peak memory usage as peak size. In order to perform the GPU/CPU data movement in the background, GeePS does double buffering by making the buffer pool twice as large as the peak size.

#### Data Placement Policy

We will now describe our policy for choosing which data to pin in GPU memory. In our implementation, any local data that is pinned in GPU memory does not need to use any access buffer space. The allocator thread will just give the pointer to the pinned GPU local data to the application, without copying the data. For the parameter data, even though it is pinned in GPU memory, the allocator thread still needs to copy it from the parameter cache to an access buffer, because the parameter cache could be modified by the background communication thread (the puller thread) while the application is doing computation. As a result, pinning local data in GPU memory gives us more benefit than pinning parameter cache. Moreover, if we pin the local data that is used at the peak usage, we can reduce the peak access buffer usage, because it doesn’t need the buffer, allowing us to reserve less memory for the access buffer.

Algorithm 3 illustrates our GPU/CPU data placement policy, and it is run at the setup stage, after the access information is gathered from the application. The algorithm chooses the entries to pin in GPU memory based on the gathered access information and a given GPU memory budget. While keeping the access buffer pool twice the peak size for double buffering, our policy will first try to pin the local data that is used at the peak in GPU memory, in order to reduce the peak size and thus the size of the buffer pool. Then, it will try to use the available capacity to pin
Algorithm 3  GPU/CPU data placement policy

**Input:** paramData, localData ← parameter data and local data accessed at each layer
**Input:** totalMem ← the amount of GPU memory to use

# Start with everything in CPU memory

cpuData ← allLocalData ∪ allParamData
gpuData ← ∅

# Set access buffer twice the peak size for double buffering

peakSize ← peak data usage, excluding GPU local data
bufferSize ← 2 × peakSize
availMem ← totalMem − bufferSize

# First pin local data used at peak

while more local data in cpuData do
    localData ← CPU local data used at peak
    Δ(peakSize) ← peakSize change if move localData to GPU
    Δ(memSize) ← size(localData) + 2 × Δ(peakSize)
    if availMem < Δ(memSize) then
        break
    end if
    Move localData from cpuData to gpuData
    availMem ← availMem − Δ(memSize)
end while

# Pin more local data using the available memory

for each localData in cpuData do
    if availMem ≥ size(localData) then
        Move localData from cpuData to gpuData
        availMem ← availMem − size(localData)
    end if
end for

# Pin parameter data using the available memory

for each paramData in cpuData do
    if availMem ≥ size(paramData) then
        Move paramData from cpuData to gpuData
        availMem ← availMem − size(paramData)
    end if
end for

# Use the reset available memory as access buffer

Increase bufferSize by availMem
more local data and parameter cache data in GPU memory. Finally, it will add the rest available GPU memory to the access buffer.

**Avoiding Unnecessary Data Movement**

When the application accesses the local data that is stored in CPU memory, usually the allocator thread will need to allocate a piece of GPU memory from the buffer pool and copy the data from CPU memory to GPU memory. However, sometimes this data movement is unnecessary, when the application just needs an uninitialized piece of GPU memory. For example, when we train a deep neural network, the input data and intermediate data will be overwritten by a new mini-batch, so the old values from the last mini-batch can be safely thrown away. To avoid the unnecessary data movement, we allow the application to specify a **no-fetch** flag when calling `LOCAL-ACCESS`, and it tells GeePS to just allocate a piece of GPU memory, without fetching the data from CPU memory. Similarly, when the application calls `POST-LOCAL-ACCESS`, it can use a **no-save** flag to tell GeePS to just free the GPU memory, without saving the data to CPU memory.

### 4.4 Evaluation

This section evaluates GeePS’s using three image classification models and a video classification model executed in the original and modified Caffe application. The evaluation confirms three main findings:

- GeePS provides effective data-parallel scaling of training throughput and training convergence rate, at least up to 16 machines with GPUs.

- GeePS’s efficiency is much higher, for GPU-based training, than a traditional CPU-based parameter server and also much higher than parallel CPU-based training performance reported in the literature.

- GeePS’s dynamic management of GPU memory allows data-parallel GPU-based training on models that are much larger than used in state-of-the-art deep learning for image classification and video classification.

A specific non-goal of our evaluation is comparing the classification accuracies of the different models. Our focus is on enabling faster training of whichever model is being used, which is why we measure performance for several.

#### 4.4.1 Experimental Setup

**Application Setup**

We use Caffe [69], the open-source single-GPU convolutional neural network application discussed earlier.¹ Our experiments use unmodified Caffe to represent the optimized single-GPU

¹For the image classification application, we used the version of Caffe from `https://github.com/BVLC/caffe` as of June 19, 2015. Since their master branch version does not support RNN, for the video classification
Algorithm 4 GeePS-Caffe training with virtual iteration

$L \leftarrow \text{number of layers in the network}$
$paramDataKeys \leftarrow \text{decide row keys for param data}$
$localDataKeys \leftarrow \text{decide row keys for local data}$

# Report access information with virtual iteration

TRAINMINIBATCH(null, virtual = yes)

# Real training iterations

while not done do
    TRAINMINIBATCH(nextTrainData, virtual = false)
end while

function TRAINMINIBATCH(trainData, virtual)

    # Forward pass
    for $i = 0 \sim (L - 1)$ do
        paramDataPtr $\leftarrow$
        geeps.Read(paramDataKeys$_i$, virtual)
        localDataPtr $\leftarrow$
        geeps.LocalAccess(localDataKeys$_i$, virtual)
        if not virtual then
            Setup layer$_i$ with data pointers
            Forward computation of layer$_i$
        end if
        geeps.PostRead(paramDataPtr)
        geeps.PostLocalAccess(localDataPtr)
    end for

    # Backward pass
    for $i = (L - 1) \sim 0$ do
        paramDataPtr $\leftarrow$
        geeps.Read(paramDataKeys$_i$, virtual)
        paramUpdatePtr $\leftarrow$
        geeps.PreWrite(paramDataKeys$_i$, virtual)
        localDataPtr $\leftarrow$
        geeps.LocalAccess(localDataKeys$_i$, virtual)
        if not virtual then
            Setup layer$_i$ with data pointers
            Backward computation of layer$_i$
        end if
        geeps.PostRead(paramDataPtr)
        geeps.Write(paramUpdatePtr)
        geeps.PostLocalAccess(localDataPtr)
        geeps.TableClock(table = $i$)
    end for
end function
case and a minimally modified instance (GeePS-Caffe) that uses GeePS for data-parallel execution. As shown in Algorithm 4, GeePS-Caffe uses GeePS to manage all its parameter data and local data (including input data and intermediate data). The parameter data of each layer is stored as rows of a distinct GeePS tables, allowing GeePS to propagate the each layer’s updates during the computations of other layers, as suggested by Zhang et al. [167]. In order to enable the specializations from access information hints, GeePS-Caffe performs a virtual iteration to report its access information to GeePS, before starting the real computations. The GeePS calls with a virtual flag are just recorded but without any further actions taken.

Cluster Setup

Each machine in our cluster has one NVIDIA Tesla K20C GPU, which has 5 GB of GPU device memory. In addition to the GPU, each machine has four 2-die 2.1 GHz 16-core AMD® Opteron 6272 packages and 128 GB of RAM. Each machine is installed with 64-bit Ubuntu 14.04, CUDA toolkit 7.5, and cuDNN v2. The machines are inter-connected via a 40 Gbps Ethernet interface (12 Gbps measured via iperf), and Caffe reads the input training data from remote file servers via a separate 1 Gbps Ethernet interface.

Image Classification Datasets and Models

The experiments use two datasets for image classification. The first one is the ImageNet22K dataset [30], which contains 14 million images labeled to 22,000 classes. Because of the computation work required to train such a large dataset, CPU-based systems running on this dataset typically need a hundred or more machines and spend over a week to reach convergence [19]. We use half of the images (7 million images) as the training set and the other half as the testing set, which is the same setup as described by Chilimbi et al. [19]. For the ImageNet22K dataset, we use a similar model to the one used to evaluate ProjectAdam [19], which we refer to as the AdamLike model. The AdamLike model has five convolutional layers and three fully connected layers. It contains 2.4 billion connections for each image, and the model parameters are 470 MB in size.

The second dataset we used is ILSVRC12 [30]. It is a subset of the ImageNet22K dataset, with 1.3 million images labeled to 1000 classes. For this dataset, we use the GoogLeNet, a recent inception model from Google. The network has about 100 layers, and 22 of them have model parameters. Though the number of layers is large, the model parameters are only 57 MB in size, because they use mostly convolutional layers.

Video Classification Datasets and Models

We use the UCF-101 dataset [133] for our video classification experiments. UCF-101 has about 8,000 training videos and 4,000 testing videos categorized into 101 human action classes. We application, we used the version from https://github.com/LisaAnne/lisa-caffe-public/tree/lstm_video_deploy as of Nov 9, 2015. We were not able to obtain the exact model that ProjectAdam uses, so we emulated it based on the descriptions in the paper. Our emulated model has the same number and types of layers and connections, and we believe our training performance evaluations are representative even if the resulting model accuracy may not be.
use a recurrent neural network model for video classification, following the approach described by Donahue et al. [33]. We use the GoogLeNet network as the CNN layers and stack LSTM layers with 256 hidden units on top of them. The weights of the GoogLeNet layers have been pre-trained with the single frames of the training videos. Following the same approach as is described by Donahue et al. [33], we extract the video frames at a rate of 30 frames per second and train the model with randomly selected video clips of 32 frames each.

**Training Algorithm Setup**

Both the unmodified and the GeePS-hosted Caffe train the models using the SGD algorithm with a momentum of 0.9. Unless otherwise specified, we use the configurations listed in Table 4.1. Our experiments in §4.4.3 evaluate performance with different mini-batch sizes. For AdamLike and GoogLeNet network, we used the same learning rate for both single-machine training and distributed training, because we empirically found that this learning rate is the best setting for both. For the RNN model, we used a different learning rate for single-machine training, because it leads to faster convergence.

<table>
<thead>
<tr>
<th>Model</th>
<th>Mini-batch size (per machine)</th>
<th>Learning rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdamLike</td>
<td>200 images</td>
<td>0.0036, div by 10 every 3 epochs</td>
</tr>
<tr>
<td>GoogLeNet</td>
<td>32 images</td>
<td>0.0036, div by 10 every 150 epochs</td>
</tr>
<tr>
<td>RNN</td>
<td>1 video, 32 frames each</td>
<td>0.0000125 for 8 machines, 0.0001 for single-machine, div by 10 every 20 epochs</td>
</tr>
</tbody>
</table>

Table 4.1: Model training configurations.

**GeePS Setup**

We run one application worker (Caffe linked with one GeePS instance) on each machine. Unless otherwise specified, we let GeePS keep the parameter cache and local data in GPU memory for each experiment, since it all fits for all of the models used; §4.4.3 evaluates performance when keeping part of the data in CPU memory, including for a very large model scenario. Unless otherwise specified, BSP mode is used.

**4.4.2 Scaling Deep Learning with GeePS**

This section evaluates how well GeePS supports data-parallel scaling of GPU-based training on both the image classification application and the video classification application. We compare **GeePS** with three classes of systems:

- **Caffe**: Single-GPU optimized training with the original unmodified Caffe system, which represents training optimized for execution on a single GPU.

- **CPU-PS**: multiple instances of the modified Caffe running on GPU workers linked via IterStore [22], a state-of-the-art CPU-based parameter server.
• **CPU workers with CPU-based parameter server**: reported performance numbers from recent literature are used to put the GPU-based performance into context relative to state-of-the-art CPU-based deep learning.

Figure 4.7: Image classification throughput scalability for (upper) AdamLike model on ImageNet22K dataset and (bottom) GoogLeNet model on ILSVRC12 dataset. Both GeePS and CPU-PS run in the fully synchronous mode.

**Image Classification**

Figure 4.7 shows model training throughput of the image classification application, in terms of both number of images trained per second and number of neural network connections trained per second. Note that there is a linear relationship between those two metrics. GeePS scales almost linearly when we add more machines. Compared to the single-machine optimized Caffe, GeePS achieves $13 \times$ speedups on both GoogLeNet and AdamLike model using 16 machines. Compared to CPU-PS, GeePS achieves over $2 \times$ more throughput.
Chilimbi et al. [19] report that ProjectAdam can train 570 billion connections per second on the ImageNet22K dataset when using 108 machines (88 CPU-based worker machines with 20 parameter server machines). Figure 4.7(upper) shows the GeePS achieves higher throughput using only 4 GPU machines, because of efficient data-parallel execution on GPUs.

Figure 4.8: Image classification top-1 accuracies for (upper) AdamLike model on ImageNet22K dataset and (bottom) GoogLeNet model on ILSVRC12 dataset.

Figure 4.8 shows the image classification top-1 testing accuracies of our trained models. To evaluate convergence speed, we compare the amount of time required to reach a given level of accuracy, which is a combination of image training throughput and model convergence per trained image. For the AdamLike model on the ImageNet22K dataset, Caffe needs 26.9 hours to reach 10% accuracy, while GeePS needs only 4.6 hours using 8 machines (6× speedup) or 3.3 hours using 16 machines (8× speedup). For the GoogLeNet model on the ILSVRC12 dataset, Caffe needs 13.7 hours to reach 30% accuracy, while GeePS needs only 2.8 hours using 8 machines (5× speedup) or 1.8 hours using 16 machines (8× speedup). The model training time speedups compared to the single-GPU optimized Caffe are lower than the image training throughput speedups, as expected, because each machine determines gradients independently. Even using BSP, more
training is needed than with a single worker to make the model converge. But, the speedups are still substantial.

For the AdamLike model on the ImageNet22K dataset, Chilimbi et al. [19] report that ProjectAdam needs one day to reach 13.6% accuracy using 58 machines (48 CPU-based worker machines with 10 parameter server machines). GeePS needs only 6 hours to reach the same accuracy using 16 machines (about $4 \times$ speedup). To reach 13.6% accuracy, the DistBelief system trained (a different model) using 2,000 machines for a week [28].

Because both GeePS and CPU-PS run in the BSP mode using the same number of machines, the accuracy improvement speedups of GeePS over CPU-PS are the same as the throughput speedups, so we leave them out of the graphs.

![Figure 4.9: Video classification task: (upper) training throughput; (bottom) top-1 accuracies.](image-url)
Video Classification

Figure 4.9(upper) shows the training throughput of the video classification application. GeePS scales linearly from Caffe (8× throughput with 8 machines). Figure 4.9(bottom) shows the top-1 testing accuracies. To reach 60% accuracy, Caffe needs 3.6 hours, while GeePS needs 0.5 hours (7× speedup); to reach 68% accuracy, Caffe needs 8.4 hours, while GeePS needs 2.4 hours (3.5× speedup).

4.4.3 Dealing with Limited GPU Memory

An oft-mentioned concern with data-parallel deep learning on GPUs is that it can only be used when the entire model, as well as all intermediate state and the input mini-batch, fit in GPU memory. GeePS eliminates this limitation with its support for managing GPU memory and using it to buffer data from the much larger CPU memory. Although all of the models we experiment with (and most state-of-the-art models) fit in our GPUs’ 5 GB memories, we demonstrate the efficacy of GeePS’s mechanisms in two ways: by using only a fraction of the GPU memory for the largest case (AdamLike) and by experimenting with a much larger synthetic model. We also show that GeePS’s memory management support allows us to do video classification on longer videos.

Artificially Shrinking Available GPU Memory

With a mini-batch of 200 images per machine, training the AdamLike model on the ImageNet22K dataset requires only 3.67 GB, with 123 MB for input data, 2.6 GB for intermediate states, and 474 MB each for parameter data and computed parameter updates. Note that the sizes of the parameter data and parameter updates are determined by the model, while the input data and intermediate states grow linearly with the mini-batch size. For best throughput, GeePS also requires use of an access buffer that is large enough to keep the actively used parameter data and parameter updates at the peak usage, which is 528 MB minimal and 1.06 GB for double buffering (the default) to maximize overlapping of data movement with computation. So, in order to keep everything in GPU memory, the GeePS-based training needs 4.73 GB of GPU memory.

Recall, however, that GeePS can manage GPU memory usage such that only the data needed for the layers being processed at a given point need to be in GPU memory. Figure 4.10 shows the per-layer memory usage for the AdamLike model training, showing that it is consistently much smaller than the total memory usage. The left Y axis shows the absolute size (in GB) for a given layer, and the right Y axis shows the fraction of the absolute size over the total size of 4.73 GB. Each bar is partitioned into the sizes of input data, intermediate states, parameter data, and parameter updates for the given layer. Most layers have little or no parameter data, and most of the memory is consumed by the intermediate states for neuron activations and error terms. The layer that consumes the most memory uses about 17% of the total memory usage, meaning that about 35% of the 4.73 GB is needed for full double buffering.

Figure 4.11 shows data-parallel training throughput using 8 machines, when we restrict GeePS to using different amounts of GPU memory to emulate GPUs with smaller memories. When there is not enough GPU memory to fit everything, GeePS must swap data to CPU mem-
Figure 4.10: Per-layer memory usage of AdamLike model on ImageNet22K dataset.

Figure 4.11: Throughput of AdamLike model on ImageNet22K dataset with different GPU memory budgets.

ory. For the case of 200 images per batch, when we swap all data in CPU memory, we need only 35% of the GPU memory compared to keeping all data in GPU memory, but we are still able to get 73% of the throughput.

When the GPU memory limits the scale, people are often forced to use smaller mini-batch sizes to let everything fit in GPU memory. Our results in Figure 4.11 also shows that using our memory management mechanism is more efficient than shrinking the mini-batch size. For the three mini-batch sizes compared, we keep the inter-machine communication the same by doing multiple mini-batches per clock as needed (e.g., four 50-image mini-batches per clock). For the case of 100 images per batch and 50 images per batch, 3.7 GB and 3.3 GB respectively are needed to keep everything in GPU memory (including access buffers for double buffering), and the rest of the available GPU memory is used to extend the access buffers. While smaller mini-batches reduce the total memory requirement, they perform significantly less well for two primary reasons: (1) the GPU computation is more efficient with a larger mini-batch size, and
(2) the time for reading and updating the parameter data locally, which does not shrink with mini-batch size, is amortized over more data.

**Training a Very Large Neural Network**

To evaluate performance for much larger neural networks, we create and train huge synthetic models. Each such neural network contains only fully connected layers with no weight sharing, so there is one model parameter (weight) for every connection. The model parameters of each layer is about 373 MB. We create multiple such layers and measure the throughput (in terms of # connections per second) of training different sized networks. Figure 4.12 shows the results. For all sizes tested, up to a 20 GB model (56 layers) that requires over 70 GB total (including local data), GeePS is able to train the neural network without excessive overhead. The overall result is that GeePS’s GPU memory management mechanisms allows data-parallel training of very large neural networks, bounded by the largest layer rather than the overall model size.

![Graph showing training throughput on very large models.](image)

Figure 4.12: Training throughput on very large models. Note that the number of connections increases linearly with model size, so the per-image training time grows with model size because the per-connection training time stays relatively constant.

**Video Classification on Longer Videos**

The video classification application requires complete sequences of image frames to be in the same mini-batch, so the GPU memory size will either limit the maximum number of frames per video or force the model to be split across multiple machines, incurring extra complexity and network communication overhead. Using unmodified Caffe, for example, our RNN can support a maximum video length of 48 frames. Because the videos are often sampled at a rate of 30 frames per second, a 48-frame video is less than 2 seconds in length. Ng et al. [165] find that using more frames in a video improves the classification accuracy. In order to use a video length of 120 frames, Ng et al. use a model parallel approach to split the model across four

---

3Here, we are just considering the memory consumption of training. If we further consider the memory consumption for testing, the supported maximum video length is even lower.
Figure 4.13: (a) A two-level category hierarchy taken from ImageNet-1000 dataset. (b) The hierarchical deep convolutional neural network (HD-CNN) architecture, where orange and yellow blocks are layers introduced in addition to the backbone model.

machines, which incurs extra network communication overhead. By contrast, with the memory management support of GeePS, we are able to train videos with up to 192 frames, with pure data parallelism.

So far, we have evaluate the efficacy of MemSwap and GeePS on standard models and tasks. Next, we show that the optimizations presented in Chapter 3 and Chapter 4 open the design space of models with extremely more parameters and memory footprints, which in turn boosts task performance.

### 4.5 Application: Hierarchical Deep Convolutional Neural Networks (HD-CNN)

In image classification, visual separability between different object categories is highly uneven, and some categories are more difficult to distinguish than others. Such difficult categories demand more dedicated classifiers. However, existing deep convolutional neural networks (CNNs) are trained as flat N-way classifiers, and few efforts have been made to leverage the hierarchical structure of categories. A straightforward idea to enhance existing models is to embed multi-level category hierarchy into CNNs, and to direct the CNN to learn higher-level features to distinguish between coarse but easier categories (e.g. cars vs. plants) first, and then more subtle and dedicate features to classify finer-grained but more difficult categories (e.g. eagle vs. hawk) [31, 127, 162].

The realization of this motivation (which is also called HD-CNN [162] shown in Figure 4.13), is however hindered by two system challenges: (1) Introducing multi-level category hierarchies significantly deepens the backbone CNN with more feature learning layers and intermediate classifiers; (2) Each added layer would generate more intermediate states that need to be stored for backpropagation, increasing peak memory usage that easily exceeds the GPU memory limit. Every intermediate classifier for coarse categories brings in at least one fully-connected softmax with dense matrix-shaped parameters, significantly burdening the network communication in distributed training.
Scaling Up HD-CNN

We incorporate a two-layered coarse-to-fine category hierarchy on ImageNet-1000 [30], using VGG-19 [129] as the build block model, resulting in 4x more parameters. We optimize the training of such a neural network (shown in Figure 4.13) using the strategies presented in the previous two chapters. We show that scaling up HD-CNN is a perfect examination of the proposed approaches.

To reduce the peak GPU memory usage, we let the system swap inactive intermediate states back to host CPU memory based on the model structure, and reserve space on GPU memory for the upcoming computation (§4.2.3). To communicate the parameters of HD-CNN, we customize the communication scheme using the adaptive communication (§3.3) based on the layer type and number of neurons, effectively minimizing the size of messages added by more fully-connected layers.

<table>
<thead>
<tr>
<th>Method</th>
<th>top-1, top-5</th>
</tr>
</thead>
<tbody>
<tr>
<td>GoogLeNet, multi-crop [137]</td>
<td>N/A, 7.9</td>
</tr>
<tr>
<td>VGG-19-layer, dense [129]</td>
<td>24.8, 7.5</td>
</tr>
<tr>
<td>VGG-16-layer + VGG-19-layer, dense</td>
<td>24.0, 7.1</td>
</tr>
<tr>
<td>Base: ImageNet-VGG-16-layer, dense</td>
<td>24.79, 7.50</td>
</tr>
<tr>
<td>HD-CNN w/ 2-layer hierarchy, dense</td>
<td>23.69, 6.76</td>
</tr>
</tbody>
</table>

Table 4.2: Errors on ImageNet validation set.

We successfully scaled out this model onto 32 GPU-nodes, each with (12Gb) standard device memory. Note that this HD-CNN has nearly 300M parameters, and would need 70Gb GPU memory to train otherwise. We reported state-of-the-art image classification performance on ImageNet back in 2015 [162] in Table 4.2.

4.6 Additional Related Work

GPU-specialized Distributed Deep Learning Systems

Deep Image [154] is a custom-built supercomputer for deep learning via GPUs. Their image classification application fit within the memory of GPUs they used, allowing use of data-parallel execution. They also support for model-parallel execution, with ideas borrowed from Krizhevsky et al. [81]. They partition the model on fully connected layers, but not on convolutional layers. The machines are interconnected by Infiniband with GPUDirect RDMA, so no CPU involvement is required, and they do not use the CPU cores or CPU memory to enhance scalability like GeePS does. Deep Image exploits its low latency GPU-direct networking for specialized parameter state exchanges rather than using a general parameter server architecture like GeePS.

MXNet [14] and Poseidon [167] are two concurrently developed systems for multi-GPU deep learning. Both systems take the data-parallel approach by making use of CPU-based parameter servers. GeePS differs from MXNet and Poseidon in two primary ways. First, in order to overcome the inefficiency of using CPU-based parameter servers, both MXNet and Poseidon rely on
optimizations to their specific GPU application system. GeePS, on the other hand, specializes its reusable parameter server module providing efficiency for all GPU deep learning applications it hosts. Indeed, the application improvements made for MXNet and Poseidon would further improve our reported performance numbers. Second, both MXNet and Poseidon require that each of their GPU machines has enough GPU memory to store all model parameters and intermediate data, limiting the size of their neural networks. GeePS’s explicit GPU memory management support, on the other hand, allows the training of neural networks that are much bigger than the available GPU memory.

Memory Management in Deep Learning

MemSwap-based.
Remateriazation-based.
Chapter 5

Consistency Model

In this chapter, we move our attention to the aspect of consistency model in distributed parallel machine learning. Most distributed machine learning (ML) systems store a copy of the model parameters locally on each machine to minimize network communication. In practice, in order to reduce synchronization waiting time, these copies of the model are not necessarily updated in lock-step, and can become stale. Despite much development in large-scale ML, the effect of staleness on the learning efficiency is inconclusive, mainly because it is challenging to control or monitor the staleness in complex distributed environments.

This chapter studies the convergence behaviors of a wide array of ML models and algorithms under delayed updates, by resorting to pure empirical simulations. Our extensive experiments reveal the rich diversity of the effects of staleness on the convergence of ML algorithms and offer insights into seemingly contradictory reports in the literature.

These conclusions confirm that in terms of consistency, different models, or even building blocks of models, have different degrees of robustness against staleness, which offers design space for future systems to adapt the consistency model (which were usually implemented as fixed in existing ML systems) to best trade-off between system throughput and statistical efficiency. Unlike previous chapters, however, we defer such a system implementation with adaptive consistency to future study.

5.1 Distributed ML: Synchronous or Asynchronous?

In §?? we have defined the basic concept of consistency model in distributed ML. In this section, we deepen the discussion and aim to address a more practical problem: when, for what models and algorithms, should one use synchronous training, and non-synchronous training. Note that we use the term non-synchronous to include both fully asynchronous model [120] and bounded asynchronous models such as Stale Synchronous Parallel [60].

With the advent of big data and complex models, there is a growing body of works on scaling machine learning under synchronous and non-synchronous distributed execution [28, 46, 87]. These works, however, point to seemingly contradictory conclusions on whether non-synchronous execution outperforms synchronous counterparts in terms of absolute convergence, which is measured by the wall clock time to reach the desired model quality. For deep neu-
ral networks, Chilimbi et al. [19] and Dean et al. [28] show that fully asynchronous systems achieve high scalability and model quality, but others argue that synchronous training converges faster [11, 23]. The disagreement goes beyond deep learning models: a body of work [60, 120, 171, 173] have empirically and theoretically show that many algorithms scale effectively under non-synchronous settings, but another line of work [53, 98, 102] demonstrate significant penalties from asynchrony.

The crux of the disagreement lies in the trade-off between two factors contributing to the absolute convergence: statistical efficiency and system throughput. Statistical efficiency measures convergence per algorithmic step (e.g., a mini-batch), while system throughput captures the performance of the underlying implementation and hardware. Non-synchronous execution can improve system throughput due to lower synchronization overheads, which is well understood [11, 19, 22, 60]. However, by allowing various workers to use stale versions of the model that do not always reflect the latest updates, non-synchronous systems can exhibit lower statistical efficiency [11, 23]. How statistical efficiency and system throughput trade off in distributed systems is far from clear.

The difficulties in understanding the trade-off arise because statistical efficiency and system throughput are coupled during execution in distributed environments. Non-synchronous executions are in general non-deterministic, which can be difficult to profile. Furthermore, large-scale experiments are sensitive to the underlying hardware and software artifacts, which confounds the comparison between studies. Even when they are controlled, innocuous change in the system configurations such as adding more machines or sharing resources with other workloads can inadvertently alter the underlying staleness levels experienced by ML algorithms, masking the true effects of staleness.

Understanding the impact of staleness on ML convergence independently from the underlying distributed systems is a crucial step towards decoupling statistical efficiency from the system complexity. The gleaned insights can also guide distributed ML system development, potentially using different synchronization for different problems. In particular, we are interested in the following aspects: Do ML algorithms converge under staleness? To what extent does staleness impact the convergence?

By resorting to simulation study, we side step the challenges faced in distributed execution. We study the impact of staleness on a diverse set of models: Convolutional Neural Networks (CNNs), recurrent neural networks (RNNs), Deep Neural Networks (DNNs), multi-class Logistic Regression (MLR), Matrix Factorization (MF), Latent Dirichlet Allocation (LDA), and Variational Autoencoders (VAEs). They are addressed by 7 algorithms, spanning across optimization, sampling, and blackbox variational inference. Our findings suggest that while some algorithms are more robust to staleness, no ML method is immune to the negative impact of staleness. We find that all investigated algorithms reach the target model quality under moderate levels of staleness, but the convergence can progress very slowly or fail under high staleness levels. The effects of staleness are also problem dependent. For CNNs, DNNs, and RNNs, the staleness slows down deeper models more than shallower counterparts. For MLR, a convex objective, staleness has minimal effect. Different algorithms respond to staleness very differently. For example, high staleness levels incur more statistical penalty for Momentum methods than stochastic gradient descent (SGD) and Adagrad [34]. Separately, Gibbs sampling for LDA is highly resistant to staleness up to a certain level, beyond which it does not converge to a fixed
point. Overall, it appears that staleness is a key governing parameter of ML convergence.

5.2 Method and Setup

We study six ML models and focus on algorithms that lend itself to data parallelism, which is a primary approach for distributed ML. Our algorithms span optimization, sampling, and black box variational inference. Table 5.1 summarizes the studied models and algorithms.

5.2.1 Simulation Model

Each update generated by worker p needs to be propagated to both worker p’s model cache and other worker’s model cache. We apply a uniformly random delay model to these updates that are in transit. Specifically, let \( u_t^p \) be the update generated at iteration \( t \) by worker \( p \). For each worker \( p' \) (including \( p \) itself), our delay model applies a delay \( r_{t,p,p'}^t \sim \text{Categorical}(0, 1, \ldots, s) \), where \( s \) is the maximum delay and \( \text{Categorical()} \) is the categorical distribution placing equal weights on each integer. Under this delay model, update \( u_t^p \) shall arrive at worker \( p' \) at the start of iteration \( t + 1 + r_{t,p,p'}^t \). The average delay under this model is \( \frac{1}{2} s + 1 \). Notice that for one worker with \( s = 0 \) we reduce to the sequential setting. Since model caches on each worker are symmetric, we use the first worker’s model to evaluate the model quality. Finally, we are most interested in measuring convergence against the logical time, and wall clock time is in general immaterial as the simulation on a single machine is not optimized for performance.

5.2.2 Models and Algorithms

Convolutional Neural Networks (CNNs)

CNNs have been a strong focus of large-scale training, both under synchronous [23, 46, 169] and non-synchronous [11, 28, 53, 93] training. We consider residual networks with \( 6n + 2 \) weight layers [57]. The networks consist of 3 groups of \( n \) residual blocks, with 16, 32, and 64 feature maps in each group, respectively, followed by a global pooling layer and a softmax layer. The residual blocks have the same construction as in [57]. We measure the model quality using test accuracy. For simplicity, we omit data augmentation in our experiments.

Deep Neural Networks (DNNs)

DNNs are neural networks composed of fully connected layers. Our DNNs have 1 to 6 hidden layers, with 256 neurons in each layer, followed by a softmax layer. We use rectified linear units (ReLU) for nonlinearity after each hidden layer [106].

Multi-class Logistic Regression (MLR)

MLR is the special case of DNN with 0 hidden layers. We measure the model quality using test accuracy.
<table>
<thead>
<tr>
<th>Model</th>
<th>Algorithms</th>
<th>Key Parameters</th>
<th>Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNN</td>
<td>SGD</td>
<td>η</td>
<td>CIFAR10 (CNN)</td>
</tr>
<tr>
<td></td>
<td>Momentum SGD</td>
<td>η, momentum=0.9</td>
<td>Penn Treebank (RNN)</td>
</tr>
<tr>
<td></td>
<td>Adam</td>
<td>η, β₁ = 0.9, β₂ = 0.999</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Adagrad</td>
<td>η</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RMSProp</td>
<td>η, decay=0.9, momentum=0</td>
<td></td>
</tr>
<tr>
<td>RNN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DNN/MLR</td>
<td>SGD</td>
<td>η = 0.01</td>
<td>MNIST</td>
</tr>
<tr>
<td></td>
<td>Adam</td>
<td>η = 0.001, β₁ = 0.9, β₂ = 0.999</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Adagrad</td>
<td>η</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RMSProp</td>
<td>η, decay=0.9, momentum=0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Adam</td>
<td>η, β₁ = 0.9, β₂ = 0.999</td>
<td></td>
</tr>
<tr>
<td>LDA</td>
<td>Gibbs Sampling</td>
<td>α = 0.1, β = 0.1</td>
<td>20 NewsGroup</td>
</tr>
<tr>
<td>MF</td>
<td>SGD</td>
<td>η = 0.005, rank=5, λ = 0.0001</td>
<td>MovieLens1M</td>
</tr>
<tr>
<td></td>
<td>Adagrad</td>
<td>η</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RMSProp</td>
<td>η, decay=0.9, momentum=0</td>
<td></td>
</tr>
<tr>
<td>VAE</td>
<td>Blackbox VI</td>
<td>Optimization parameters same as MLR/DNN</td>
<td>MNIST</td>
</tr>
</tbody>
</table>

Table 5.1: Overview of the models, algorithms [34, 49, 77, 121], and dataset in our study. η denotes learning rate, which, if not specified, are tuned empirically for each algorithm and staleness level (over η = 0.001, 0.01, 0.1), β₁, β₂ are optimization hyperparameters (using common default values). α, β in LDA are Dirichlet priors for document topic and word topic random variables, respectively.

Matrix Factorization (MF)

MF is commonly used in recommender systems and have been implemented at scale [22, 60, 74, 83, 89, 95]. Let \( D \in \mathbb{R}^{M \times N} \) be a partially filled matrix, MF factorizes \( D \) into two factor matrices \( L \in \mathbb{R}^{M \times r} \) and \( R \in \mathbb{R}^{N \times r} \) (\( r < \min(M,N) \) is the user-defined rank). The \( \ell_2 \)-penalized optimization problem is

\[
\min_{L,R} \frac{1}{|D_{obs}|} \left\{ \sum_{(i,j) \in D_{obs}} \left| D_{ij} - \sum_{k=1}^{K} L_{ik} R_{kj} \right|^2 + \lambda (||L||^2_F + ||R||^2_F) \right\},
\]

where \( || \cdot ||_F \) is the Frobenius norm and \( \lambda \) is the regularization parameter. We partition observations \( D \) to workers while treating \( L, R \) as shared model parameters. We optimize MF via SGD, and measure model quality by training loss defined by the objective function above.

Latent Dirichlet Allocation (LDA)

LDA is an unsupervised method to uncover hidden semantics (“topics”) from a group of documents, each represented as a bag of tokens. LDA has been scaled under non-synchronous execution [4, 95, 164] with great success. Further details are provided in Appendix ??.
Variational Autoencoder (VAE)

VAE is commonly optimized by black box variational inference, which can be considered as a hybrid of optimization and sampling methods. The inputs to VAE training include two sources of stochasticity: the data sampling $x$ and samples of random variable $\epsilon$. We measure the model quality by test loss. We use DNNs with 1~3 layers as the encoders and decoders in VAE, in which each layer has 256 units furnished with rectified linear function for non-linearity. The model quality is measured by the training objective value, assuming continuous input $x$ and isotropic Gaussian prior $p(z) \sim \mathcal{N}(0, I)$.

![Figure 5.1](image)

**Figure 5.1:** (a)(c) The number of batches to reach 71% test accuracy on CIFAR10 for 4 variants of ResNet with varying staleness, using 8 workers and SGD (learning rate 0.01) and Adam (learning rate 0.001). The mean and standard deviation are calculated over 3 randomized runs. (b)(d) The same metrics as (a)(c), but each model is normalized by the value under staleness $0 (s = 0)$, respectively. (e)(f) The number of batches to reach 92% accuracy for MLR and DNN with varying depths, normalized by the value under staleness 0. MLR with SGD does not converge within the experiment horizon (77824 batches) and thus is omitted in (f).

### 5.3 Empirical Findings

We use batch size 32 for CNNs, DNNs, MLR, and VAEs. For MF, we use batch size of 25000 samples, which is 2.5% of the MovieLens dataset (1M samples). We study staleness up to $s = 50$ on 8 workers, which means model caches can miss updates up to 8.75 data passes. For LDA we use $\frac{D}{10P}$ as the batch size, where $D$ is the number of documents and $P$ is the number of workers. We study staleness up to $s = 20$, which means model caches can miss updates up to 2 data passes. We measure the logical time in terms of the amount of work performed, such as the number of batches processed.
5.3.1 Convergence Slowdown

Perhaps the most prominent effect of staleness on ML algorithms is the slowdown in convergence, evident throughout the experiments. Figure 5.1 shows the number of batches needed to reach the desired model quality for CNNs and DNNs/MLR with varying network depths and different staleness ($s = 0, \ldots, 16$). Figure 5.1(b)(d) show that convergence under higher level of staleness requires more batches to be processed in order to reach the same model quality. This additional work can potentially be quite substantial, such as in Figure 5.1(d) where it takes up to 6x more batches compared with settings without staleness ($s = 0$). It is also worth pointing out that while there can be a substantial slowdown in convergence, the optimization still reaches desirable models under most cases in our experiments. When staleness is geometrically distributed and in the presence of stragglers (Figure 5.1(c)), we observe similar patterns of convergence slowdown.

We are not aware of any prior work reporting slowdown as high as observed here. This finding has important ramifications for distributed ML. Usually, the moderate amount of workload increases due to parallelization errors can be compensated by the additional computation resources and higher system throughput in the distributed execution. However, it may be difficult to justify spending large amount of resources for a distributed implementation if the statistical penalty is too high, which should be avoided (e.g., by staleness minimization system designs or synchronous execution).

5.3.2 Model Complexity

Figure 5.1 also reveals that the impact of staleness can depend on ML parameters, such as the depths of the networks. Overall we observe that staleness impacts deeper networks more than shallower ones. This holds true for SGD, Adam, Momentum, RMSProp, Adagrad (Figure 5.1), and other optimization schemes, and generalizes to other numbers of workers (see Appendix)

This is perhaps not surprising, given the fact that deeper models pose more optimization challenges even under the sequential settings [41, 57], though we point out that existing literature does not explicitly consider model complexity as a factor in distributed ML [46, 89]. Our results suggest that the staleness level acceptable in distributed training can depend strongly on the complexity of the model. For sufficiently complex models it may be more advantageous to eliminate staleness altogether and use synchronous training.

5.3.3 Algorithms’ Sensitivity to Staleness

Staleness has uneven impacts on different SGD variants. Figure 5.2 shows the amount of work (measured in the number of batches) to reach the desired model quality for five SGD variants. Figure 5.2(d)(e)(f) reveals that while staleness generally increases the number of batches needed

$^{1}$ResNet8 takes more batches to reach the same model quality than deeper networks in Figure 5.1(a) because, with SGD, ResNet8’s final test accuracy is about 73% in our setting, while ResNet20’s final test accuracy is close to 75%. Therefore, deeper ResNet can reach the same model accuracy in the earlier part of the optimization path, resulting in lower number of batches in Figure 5.1(a). However, when the convergence time is normalized by the non-stale ($s=0$) value in Figure 5.1(b), we observe the impact of staleness is higher on deeper models.
to reach the target test accuracy, the increase can be higher for certain algorithms, such as Mo-
momentum. On the other hand, Adagrad appear to be robust to staleness\(^2\). Our finding is consistent
with the fact that, to our knowledge, all existing successful cases applying non-synchronous train-
ing to deep neural networks use SGD [19, 28]. In contrast, works reporting subpar performance
from non-synchronous training often use momentum, such as RMSProp with momentum [11]
and Momentum [23]. Our results suggest that these different outcomes may be partly driven by
the choice of optimization algorithms, leading to the seemingly contradictory reports of whether
non-synchronous execution is advantageous over synchronous ones.

Effects of More Workers

The impact of staleness is amplified by the number of workers. In the case of MF, Figure 5.3(b)
shows that the convergence slowdown in terms of the number of batches (normalized by the
convergence for \(s = 0\)) on 8 workers is more than twice of the slowdown on 4 workers. For
example, in Figure 5.3(b) the slowdown at \(s = 15\) is \(\sim 3.4\), but the slowdown at the same staleness
level on 8 workers is \(\sim 8.2\). Similar observations can be made for CNNs (Figure 5.3). This can
be explained by the fact that additional workers amplifies the effect of staleness by (1) generating
updates that will be subject to delays, and (2) missing updates from other workers that are subject
to delays.

LDA

Figure 5.3(c)(d) show the convergence curves of LDA with different staleness levels for two
settings varying on the number of workers and topics. Unlike the convergence curves for SGD-
based algorithms (see Appendix), the convergence curves of Gibbs sampling are highly smooth,
even under high staleness and a large number of workers. This can be attributed to the structure
of log likelihood objective function [49]. Since in each sampling step we only update the count
statistics based on a portion of the corpus, the objective value will generally change smoothly.

Staleness levels under a certain threshold (\(s \leq 10\)) lead to convergence, following indistin-
guishable log likelihood trajectories, regardless of the number of topics (\(K = 10, 100\)) or the
number of workers (2–16 workers, see Appendix). Also, there is very minimal variance in those
trajectories. However, for staleness beyond a certain level (\(s \geq 15\)), Gibbs sampling does not
converge to a fixed point. The convergence trajectories are distinct and are sensitive to the num-
ber of topics and the number of workers. There appears to be a “phase transition” at a certain
staleness level that creates two distinct phases of convergence behaviors\(^3\). We believe this is the
first report of a staleness-induced failure case for LDA Gibbs sampling.

VAE

In Figure 5.3(e)(f), VAEs exhibit a much higher sensitivity to staleness compared with DNNs
(Figure 5.1(e)(f)). This is the case even considering that VAE with depth 3 has 6 weight layers,

\(^2\)Many synchronous systems uses batch size linear in the number of workers (e.g., [46]). We preserve the same
batch size and more workers simply makes more updates in each iteration.

\(^3\)We leave the investigation into this distinct phenomenon as future work.
which has a comparable number of model parameters and network architecture to DNNs with 6 layers. We hypothesize that this is caused by the additional source of stochasticity from the sampling procedure, in addition to the data sampling process.

![Graph](image)

Figure 5.2: (a)(b)(c) The number of batches to reach 71% test accuracy on 1, 8, 16 workers with staleness $s = 0, ..., 16$ using ResNet8. We consider 5 variants of SGD: SGD, Adam, Momentum, RMSProp, and Adagrad. For each staleness level, algorithm, and the number of workers, we choose the learning rate with the fastest time to 71% accuracy from $\{0.001, 0.01, 0.1\}$. (d)(e)(f) show the same metric but each algorithm is normalized by the value under staleness 0 ($s = 0$), respectively, with possibly different learning rate.

## 5.4 Discussion

Through the empirical studies, we show that staleness appears to be a key governing parameter in learning. Overall staleness slows down the convergence, and under high staleness levels the convergence can progress very slowly or fail. The effects of staleness are highly problem dependent, influenced by model complexity, choice of the algorithms, the number of workers, and the model itself, among others.

The findings have clear implications for distributed ML. To achieve actual speed-up in absolute convergence, any distributed ML system needs to overcome the slowdown from staleness, and carefully trade off between system throughput gains and statistical penalties. Many ML methods indeed demonstrate certain robustness against low staleness, which should offer opportunities for system optimization.
Figure 5.3: (a) The number of batches to reach training loss of 0.5 for Matrix Factorization (MF). (b) shows the same metric in (a) but normalized by the values of staleness $s$ of each worker setting, respectively (4 and 8 workers). (c)(d) Convergence of LDA log likelihood using 10 and 100 topics under staleness levels $s = 0, ..., 20$, with 2 and 16 workers. The convergence is recorded against the number of documents processed by Gibbs sampling. The shaded regions are 1 standard deviation around the means (solid lines) based on 5 randomized runs. (e)(f) The number of batches to reach test loss 130 by Variational Autoencoders (VAEs) on 1 worker, under staleness $s = 0, ..., 16$. We consider VAEs with depth 1, 2, and 3 (the number of layers in the encoder and the decoder networks, separately). The numbers of batches are normalized by $s = 0$ for each VAE depth, respectively. Configurations that do not converge to the desired test loss are omitted in the graph, such as Adam optimization for VAE with depth 3 and $s = 16$.

5.5 Additional Related Work

Staleness is reported to help absolute convergence for distributed deep learning in [19, 28, 157] and has minimal impact on convergence [53, 89, 158]. But Chen et al. and Cui et al. [11, 23] show significant negative effects of staleness. LDA training is generally insensitive to staleness [4, 60, 153, 164], and so is MF training [22, 95, 171]. However, none of their evaluations quantifies the level of staleness in the systems. By explicitly controlling the staleness, we decouple the distributed execution, which is hard to control, from ML convergence outcomes.

We focus on algorithms that are commonly used in large-scale optimization [11, 28, 46], instead of methods specifically designed to minimize synchronization [73, 108, 123]. Non-synchronous execution has theoretical underpinning [60, 88, 89, 120, 171]. Here we study algorithms that do not necessarily satisfy assumptions in their analyses.
Part II

Composability and Representations of ML Parallelisms
Overview

In the first part of the thesis, we have studied several key parallelization aspects including scheduling, communication, memory management, and consistency model, and developed aspect-specific optimizations based on the key methodology *adaptive parallelism*. This part of the thesis aims to generalize this approach – instead of developing system or algorithm optimizations at an aspect-by-aspect basis, we develop new representations to more effectively express the entirety of ML parallelization. We believe that a unified representation of parallelisms offers at least two outstanding benefits (as we will show later): improved programmability in writing parallel ML code, and opportunities of co-optimizing multiple parallelization aspects to further improve parallelization performance.

This part of thesis is organized into two chapters with respect to two distinct classes of parallel ML workloads: parallelization of dynamic DL models, which is commonly termed as *dynamic batching*, and distributed parallelization as done in previous chapters.

In particular, Chapter 6 develops new representations for expressing the *dynamic batching* parallelism, commonly found difficult but necessary for the training or inference for DL models with dynamic-varying structures. It presents two programming models developed progressively: dynamic declaration (§6.2.2), and vertex-centric representations (§6.4), and their corresponding system implementation, DyNet (§6.3) and Cavs (§6.5). DyNet and Cavs have advanced the performance of the parallelization of dynamic NNs by nearly an order of magnitude.

Chapter 7 gleans insight from our past developments on distributed parallelization of ML training, and offers a unified, principled representation (§??) to encapsulate various forms of (seemingly different) distributed parallelization strategies or optimizations. On top of the representation, it then presents a system, AutoDist (§??), that allows complex parallelization strategies being composed from base atomic parallelization aspects. AutoDist simplifies the way of writing distributed parallel ML code, with matched or even better performance than a diverse set of specialized systems, but using a unified interface. It also laids the foundation for the automatic parallelization, which will be develop in the last part of this thesis.

The results presented in this part of the thesis have appeared in the following publications:


- Hao Zhang, Christy Li, Zhijie Deng, Aurick Qiao, Qirong Ho, and Eric P. Xing. AutoDist:
A Composable and Automated Synchronization System for Distributed Deep Learning. 

*Preprint 2020.*
Chapter 6
Dynamic Neural Networks Parallelization

Recent deep learning (DL) models have moved more and more from static network architectures to dynamic ones, handling data where the network structure changes every example, such as sequences of variable lengths, trees, and graphs. However, existing DL programming models are inefficient in handling dynamic network architectures because of:

- Substantial overheads caused by repeating dataflow graph construction and processing for every input data example;
- Difficulties in batched parallel execution of multiple samples;
- Inability to incorporate graph optimization techniques such as those used in static graphs.

In this chapter, we co-develop representations and systems for dynamic neural network parallelisms. We start by giving a former introduction of dynamic neural networks, and existing programming models for such models.

6.1 Dynamic Neural Networks

Successful NN models generally exhibit suitable architectures that capture the structures of the input data. For example, convolutional neural networks [82, 162], which apply fixed-structured operations to fixed-sized images, are highly effective precisely because they capture the spatial invariance common in computer vision domains [130, 138]. However, apart from images, many forms of data are structurally complex and can not be readily captured by fixed-structured NNs. Appropriately reflecting these structures in the NN design has shown effective in sentiment analysis [140], semantic similarity between sentence pairs [132], and image segmentation [90].

To see this, we will take the constituency parsing problem as an example. Sentences in natural languages are often represented by their constituency parse tree [140], whose structure varies depending on the content of the sentence itself (Figure 6.1(a)). Constituency parsing is an important problem in natural language processing that aims to determine the corresponding grammar type of all internal nodes given the parsing tree of a sentence. Figure 6.1(b) shows an example of a network that takes into account this syntactic structure, generating representations for the sentence by traversing the parse tree bottom-up and combining the representations for each sub-
Figure 6.1: An example of a dynamic NN: (a) a constituency parsing tree, (b) the corresponding Tree-LSTM network. We use the following abbreviations in (a): S for sentence, N for noun, VP for verb phrase, NP for noun phrase, D for determiner, and V for verb.

The important observation is that the NN structure varies with the underlying parsing tree over each input sample, but the same LSTM cell is constant in shape and repeated at each internal node. Similar examples can be found for graph input [90, 91] and sequences of variable lengths [5, 136]. We refer to these NNs that exhibit different structures for different input samples as dynamic neural networks, in contrast to the static networks that have fixed network architecture for all samples. We next introduce the most adopted dynamic NNs – recurrent and recursive neural networks (RNNs) [37, 61, 132].

6.1.1 Recurrent and Recursive Neural Networks (RNNs)

RNNs are a class of NNs generally applied on modeling structured inputs or outputs, e.g., sequences or graphs. At the core of an RNN is a cell function with trainable parameters. It will be dynamically applied at different places of the input structure, and optionally produce an output if needed. Figure 6.2(a) illustrates such a cell function: it takes an input element \( x \), forwards it through a few mathematical transformations, and generates some intermediate state \( h \) and an output \( o \). Depending on what transformations are applied, different variants of RNNs have been derived, such as long-short term memory units (LSTM) [61] and gated recurrent units (GRU) [20]. However, the internals of the cells themselves are secondary; the dynamics of the net as a whole are mainly reflected by the structures that the NN works on.
Figure 6.2: A cell function shown in (a) could be applied on different structures such as a (b) chain (c) tree, or (d) graph.

Sequence RNNs

When the input to the RNN are sequences (e.g., sentences) as in Figure 6.2b, the cell function is applied across all elements of the sequence. At each step $t$, it takes the element $x_t$ (e.g., a word) from the input sequence, and the state variable $h_{t-1}$ maintained by the model at step $t-1$. It computes an output $o_t$, and a new state $h_t$ that will be used by the next step $t+1$. Hence, this sequence RNN encodes not only the data values, but also the dependencies present in the sequence. If represented as a dataflow graph, the graph exhibits a chain structure. As the input or output sequences usually have variable length (e.g., translating an arbitrary-length English sentence into Chinese), the dataflow graph needs to be dynamically changed, i.e., the steps of the chain must be adapted to fit the length of the input or output.

Tree-structured RNNs

Further, RNNs can be enhanced to model data with more complex structures suited for downstream tasks. For example, tree-structured RNNs (Tree-RNNs, Figure 6.2c), have been used to classify the sentiment of sentences [113] given an associated binary tree representing the sentence structure [131, 140]. In this case, a leaf of the tree maps to a word of the sentence, an internal node corresponds to a multi-word phrase. To process this structure, the cell function scans the tree recursively, starting from leaf nodes until reaching the root. At the node $t$, it computes the state $h_t = f(h_{tl}, h_{tr}, x_t)$, where $x_t$ is the input to the node, and $h_{tl}, h_{tr}$ are the states of its left and right children, respectively. As the tree structure vary from example to example, the dataflow graph corresponded to a Tree-RNN is highly dynamic.
Graph-structured RNNs

Similarly, RNNs can be extended to compute over more general graphs, such as N-ary trees or graphs (Figure 6.2d), as long as their parameters are learnable. In fact, various NNs have been developed toward having more dynamic workflows [90, 140], and proven quite effective because of their ability to encode structured information. While we take RNNs as examples for explanation, we note there are many other dynamic NNs in the literature or production with their dynamics reflected in various perspectives: variably sized inputs/outputs [5, 8, 35, 37, 61, 136], variably structured inputs/outputs [90, 131, 140], or with nontrivial inference algorithms [45, 48, 78, 172].

6.2 Programming and Parallelizing Dynamic Neural Networks

6.2.1 Static Declaration

There is a natural connection between NNs and directed graphs: we can map the graph nodes to the computational operations or parameters in NNs, and let the edges indicate the direction of the data being passed between the nodes. In this case, we can represent the process of training NNs as batches of data flowing through computational graphs, i.e. dataflow graphs [3, 7, 109], as we have introduced in chapter ??.

Based on dataflow graphs, one dominant paradigm in the training of DL models, adopted by almost all toolkits (TensorFlow, MxNet), is static declaration [3, 105]. In static declaration, the declared graph represents the flow of data through computational functions, and are defined using symbolic programming [3, 7], once before beginning training or testing of the model. The training of these models is performed through auto-differentiation, in which users are only required to assemble their model architectures by connecting operators using high-level language interface (e.g. Python), after which the framework will automatically derive the correct algorithm for training [6]. With proper optimization, the execution of these static dataflow graphs can be highly efficient. Specifically, by separating model declaration and execution, it makes it possible for the graph to be further processed and optimized before runtime [3]. In addition, the evaluation of multiple data samples in a dataflow graph can be naturally batched (i.e. parallelized) to leverage the improved computational capability of modern hardware (e.g. GPUs), which is extremely advantageous for DL workloads [82].

```cpp
/* static declaration */
// all samples must share one graph
declare a static dataflow graph D.
for p = 1 → P:
read the pth data batch \{x^p_k\}_{k=1}^K.
batched computation: D(\{x^p_k\}_{k=1}^K).
```

Figure 6.3: The workflows of static declaration. Notations: D notates both the dataflow graph itself and the computational function implied by it; p is the index of a batch while k is the index of a sample in the batch.
Figure 6.3 summarizes its workflow, which assumes all data samples share a fixed NN structure declared symbolically in a dataflow graph $D$. While these static dataflow graph have major efficiency advantages, their applicability highly relies on a key assumption – the dataflow graph (i.e. NN architecture) fixed throughout the runtime. Hence, in its original form, static declaration cannot express dynamic NNs with structures changing with data samples.

Two alternative forms of static declaration are derived to mitigate this issue.

**Static Unrolling**

Static unrolling [3] is a standard way to express sequence RNNs with fixed steps. To handle variable-length data, it declares an RNN that has number of steps equal with the length of the longest sequence in the dataset. It then appends zeros at the end of other sequences to have equal length, and feeds them in batches to the dataflow graph for computation. Static unrolling enables batched computation of multiple sequences, but obviously results in substantial unnecessary computation.\(^1\)

**Dynamic Unrolling**

Dynamic unrolling implements basic control flow functionality within static graphs, allowing for the declaration of graph operators similar to `while` loops. At each iteration of the training, the cell function of the RNN will be executed a conditional number of times determined at runtime by the length of the longest sentence in the batch. It then pads the sequences in the batch and performs batched computation – apparently it also waste computational resources.

Notably, both of these methods essentially cannot support more complex structures than sequences.

There are currently two remedies to this problem: expanding the graph programming language to allow it to explicitly include controls structure necessary to implement these applications, or forgo the efficiency gains afforded by static dataflow graphs and instead use a dynamic declaration framework that reconstructs the graph for every training example – the latter approach and the resultant open source framework, DyNet (to which this thesis has contributed to), are discussed in the next sections.

### 6.2.2 Dynamic Declaration

With a slight modification on static declaration, we can derive a more flexible programming model, dynamic declaration, illustrated at Figure 6.4.

By creating a unique dataflow graph $D^p_k$ for each sample $x^p_k$ according to its associated structure, dynamic declaration is able to express sample-dependent dataflow graphs. It however causes extra overhead on graph construction and puts constraints on runtime optimization, which usually lead to inefficient execution. Particularly, since a dataflow graph $D^p_k$ needs to be constructed per sample, the overhead is linearly increasing with the number of samples, and sometimes yields downgraded performance [94] (§6.6.3). Moreover, we can hardly benefit from

\(^1\)It is also possible to split sentences into several buckets of different lengths, which alleviates this problem somewhat but adds some degree of code complexity and is not a fundamental solution.
Figure 6.4: The workflow of dynamic declaration.

any well-established dataflow graph optimization (§6.5.4). We will have to perform graph processing/optimization for each dataflow graph and every single sample; but incorporating this optimization itself has a non-negligible overhead. More importantly, as we are unable to batch the computation of different structured graphs, we note in Figure 6.4 single-instance computation $D_{p k}^p(x_{p k}^k)$ would be very inefficient due to the absence of batched computation (hence underutilized GPUs).

### 6.3 DyNet: Dynamic Neural Network Toolkit

We next describe a specialized, open-sourced ML framework, DyNet, which implements dynamic declaration, and partially addresses some of the aforementioned problems (but not all). While the thesis author was an active contributor to DyNet design and code, he did not participate in the scientific studies involving benchmarking the performance of DyNet. Therefore, this section mainly aims to reveal the two key ingredients adopted in DyNet, by quoting from a white paper provided by all DyNet contributors [109], but skips a thorough evaluation of it.

#### 6.3.1 DyNet Design Overview

In contrast to the two-step process of definition and execution used by the static declaration paradigm, DyNet is built on top of the dynamic declaration model, which takes a single-step approach, in which the user defines the computation graph programmatically as if they were calculating the outputs of their network on a particular training instance. For example, in the case of image processing, this would mean that for every training example, the user would “Load a 64x64 image into their computation graph, perform several convolutions, and calculate either the predictive probabilities (for test) or loss (for training).” Notably, there are no separate steps for definition and execution: the necessary computation graph is created, on the fly, as the loss calculation is executed, and a new graph is created for each training instance. This requires very lightweight graph construction.

This general design philosophy is advantageous because it allows the user to: (1) define a different computation architecture for each training example or batch, allowing for the handling of variably sized or structured inputs using flow-control facilities of the host language, and (2) interleave definition and execution of computation, allowing for the handling of cases where the structure of computation may change depending on the results of previous computation steps. Furthermore, it reduces the complexity of the computation graph implementation since it does
not need to contain flow control operations or support dynamically sized data—these are handled by the host language (C++ or Python).

To alleviate the linearly increasing graph construction overhead and lack of batching capability, DyNet deliberately incorporate two optimizations: efficient graph construction, and on-the-fly dynamic batching.

### 6.3.2 Efficient Graph Construction

DyNet’s backend is designed with this rapid computation graph construction in mind. First and foremost, the backend is written in C++ with a specific focus on ensuring that graph building operations are as efficient as possible. Specifically, one of the major features of DyNet is that it performs its own memory management for all computation results\(^2\). When the DyNet library is initialized, it allocates three large blocks of memory, the one for storing results of forward calculation through the graph, one for storing backward calculations, and one for storing parameters and the corresponding gradients. When starting processing of a new graph, the pointers to the segments of memory responsible for forward and backward computation are reset to the top of the corresponding segments, and any previously used backward memory is zeroed out with a single operation. When more memory is required during forward execution or in advance of running the backward algorithm, the custom allocator simply returns the current pointer, then increments it according to the size of the memory segment requested. Thus, in this extremely simple memory allocation setup, deallocating and allocation of memory are simple integer arithmetic. This is also an advantage when performing operations on the GPU, as managing memory requires no interaction with the GPU itself, and can be handled entirely by managing, on the CPU, pointers into GPU memory.

Interacting with the DyNet library is also efficient. When writing programs in C++, it is possible to interact directly on the machine code level. The Python wrapper is also written with efficiency in mind, attempting to keep the wrapping code as minimal as possible, delegating the operations directly to the C++ core when possible, writing the glue code in Cython rather than Python, and trying to minimize Python objects allocation. At the same time, we also attempt to provide a “Pythonic” and natural high-level API. There is empirical evident that DyNet Python speed is indeed very close to the C++ one.

### 6.3.3 On-the-fly Dynamic Batching

The execution of dynamic NNs cannot be trivially parallelized over a batch of training data (contrast to static NNs and static declaration does), due to the ever-changing computational patterns with data points. DyNet proposes on-the-fly dynamic batching [110] to slightly alleviate this issue, which batches the operations in multiple different computational graphs with two steps.

\footnote{The data structure used to represent the computation graph is managed by the standard C++ memory management, care is taken to avoid performing expensive allocations in the construction of this structure as well.}
Checking Computing Compatibility Groups

It first partitions operations in different graphs into different groups. Operations in the same group have the potential to be batched together. The partitioning is done by creating, for each operation, a string signature expressing the input, output, the operation type, and other computation-critical information, and then grouping operations that share similar signatures.

Determining Execution Order

With the signatures, it selects an execution order to batch parallelizable operations. This execution order needs to satisfy at least two requirements: (1) an operation must be executed only when all its inputs are ready, i.e. after all its dependent operations in the graph have been executed. (2) Only nodes in the same group, without dependencies between each other, are deemed as parallelizable. Given multiple computational graphs and operations therein, finding execution order within these constraints is NP-hard. DyNet develops depth-based batching and agenda-based batching as heuristics to greedily generate sub-optimal orders.

Finally, this non-trivial batching procedure must be executed quickly so that overhead due to batch scheduling calculations does not cancel out the efficiency gains from operation batching. Empirically, this dynamic batching mechanism, implemented on top of DyNet, gives healthy improvements in computation time across a series of notable challenging-to-batch models, including BiLSTM, TreeLSTM, and Transition-Parsing: 3.6x–9.2x on the CPU, and 2.7–8.6x on GPU.

6.4 Vertex-centric Representation

While dynamic declaration is convenient to developers as it removes the restriction that computation be completely specified before training begins, it exhibits a few limitations. First, constructing a graph for every sample results in substantial overhead, which grows linearly with the number of input instances. In fact, we find graph construction takes longer time than the computation in some frameworks (see §6.6.3); even for frameworks with optimized graph construction implementations, such as DyNet, the overhead is still substantial. It also prevents the application of complex static graph optimization techniques (see §6.5.4). Moreover, since each sample owns a dataflow graph specifying its unique computational pattern, batching together similarly shaped computations across instances is non-trivial. Without parallelization, the computation is inefficient due to its lack of ability to exploit modern computational hardware. While some progress has been made in recent research [94, 110], such as the on-the-fly dynamic batching introduced in §6.3.3, how to automatically batch the computational operations from different graphs remains a difficult problem.

To fundamentally address these challenges, we present a vertex-centric programming model, and an efficient runtime system Cavs (§6.5), for dynamic NNs. Instead of declaring a dataflow graph per sample, the representation decomposes a dynamic NN into two components: a static vertex function $F$ that is only declared (by the user) and optimized once before execution, and an input-specific graph $G$ obtained via I/O at runtime. Thus, it inherits the flexibility of symbolic programming [3, 39, 109] for DL; it requires users to define $F$ by writing symbolic expressions
in the same way as in static declaration. With \( F \) and \( G \), the workflow of training or testing a dynamic NN is cast as scheduling the execution of \( F \) following the structure of the input graph \( G \). Cavs will perform auto-differentiation, schedule the execution following dependencies in \( G \), and guarantee efficiency and correctness.

We next describe the design of the vertex-centric representation and the system Cavs in detail.

### 6.4.1 Vertex-centric Programming Model

Our motivation comes from a key property of dynamic NNs: most dynamic NNs are designed to exhibit a recursive structure; Within the recursive structure, a static computational function is being applied following the topological order over instance-specific graphs. For instance, if we denote the constituency parsing tree in Figure 6.1 as a graph \( G \), where each node of the tree maps to a vertex in \( G \), we note the Tree-LSTM can be interpreted as follows: a computational cell function, specified in advance, is applied from leaves to the root, following the dependencies in \( G \). \( G \) might change with input samples, but the cell function itself is always static: It is parametrized by a fixed set of learnable parameters and interacts in the same way with its neighbors when applied at different vertices of \( G \).

These observations motivate us to decompose a dynamic NN into two parts: (1) a static computational *vertex function* \( F \) that needs to be declared by the programmer once before runtime; (2) a dynamic *input graph* \( G \) that changes with every input sample\(^3\). With this representation, the workflow of training a dynamic NN can be cast as scheduling the evaluation of the symbolic construct encoded by \( F \), following the graph dependencies of \( G \), as illustrated in Figure 6.5.

```plaintext
/* Vertex-centric model */
declare a symbolic vertex function \( F \).
for \( p = 1 \rightarrow P \):
  read the \( p \)th data batch \( \{ x^p_k \}_{k=1}^K \).
  read their associated graphs \( \{ G^p_k \}_{k=1}^K \).
  compute \( F \) over \( \{ G^p_k \}_{k=1}^K \) with inputs \( \{ x^p_k \}_{k=1}^K \).
```

Figure 6.5: The workflows of the vertex-centric programming model.

By exploiting the property of dynamic NNs, We argue that this representation addresses the aforementioned issues in the following ways.

### Minimize Graph Construction Overheads

It only requires users to declare \( F \) using symbolic expressions, and construct it once before execution. This bypasses repeated construction of multiple dataflow graphs, avoiding overheads. While it is still necessary to create an I/O function to read input graphs \( G \) for each sample, this must be done by any method, and only once before training commences, and it can be shared across samples.

\(^3\)In the following text, we will distinguish the term *vertex* from *node*. We use *vertex* to denote a vertex in the input graph while *node* to denote an operator/variable in a dataflow graph. Hence, a vertex function can have many nodes as itself represents a dataflow graph.
**Batched Execution**

With the proposed representation, Cavs transforms the problem of evaluating data samples \( \{x^p_k\}_{k=1}^K \) (at the \( p \)th batch) on different dataflow graphs \( \{D^p_k\}_{k=1}^K \) into a simpler form — scheduling the execution of the vertex function \( F \) following the dependencies in input graphs \( \{G^p_k\}_{k=1}^K \). For the latter problem, we can easily batch the execution of \( F \) on multiple vertices at runtime (§6.5.1), leveraging the batched computational capability of modern hardware and libraries.

**Open to Graph Optimizations**

Since the vertex function \( F \) encodes a dataflow graph which is static throughout runtime, it can benefit from various graph optimizations originally developed for static declaration, such as kernel fusion, streaming, and our proposed lazy batching, which are not effective in dynamic declaration (see §6.5.4).

Based on this motivation, we next describe the programming interfaces and the Cavs system. We highlight the following design challenges: (1) how to design minimal APIs in addition to the symbolic programming interface to minimize user code; (2) how to schedule the execution of \( F \) over multiple input graphs to enable batched computation; (3) how to manage memory to support the dynamic batching; (4) how to incorporate static graph optimization in Cavs’s execution engine to exploit more parallelism.

### 6.4.2 Programming Interface

Conventional dataflow graph-based programming models usually entangle the computational workflow in \( F \) with the structure in \( G \), and require users to express them as a whole in a single dataflow graph. Instead, Cavs separates the static vertex function \( F \) from the input graph \( G \) (see Fig 6.6). While users use the same set of symbolic operators [3, 38] to assemble the computational workflow in \( F \), Cavs proposes four additional APIs, \texttt{gather}, \texttt{scatter}, \texttt{pull}, \texttt{push}, to specify how the messages shall be passed between connected vertices in \( G \):

- \texttt{gather(child_idx)}: \texttt{gather} accepts an index of a child vertex, gets its output, and returns a list of symbols that represent the output of the child.

- \texttt{scatter(op)}: \texttt{scatter} reverses \texttt{gather}. It sets the output of the current vertex as \texttt{op}. If this vertex is \texttt{gathered}, the content of \texttt{op} will be returned.

\texttt{gather} and \texttt{scatter} are motivated by the GAS model in graph computing [42] — both are vertex-centric APIs that help users express the overall computational patterns by thinking locally like a vertex: \texttt{gather} receives messages from dependent vertices, while \texttt{scatter} updates information to parent vertices (see discussion in §6.7).

However, unlike graph computing, in dynamic NNs, the vertex function \( F \) usually takes input from not only the internal vertices of \( G \) (internal data path in Figure 6.6), but also the external environment, e.g. an RNN can take inputs from a CNN feature extractor or some external I/O (external data path in Figure 6.6). Cavs therefore provides another two APIs to express such semantics:
• **pull()**: pull grabs inputs from the external of the current dynamic structure, e.g. another NN, or I/O.

• **push(op)**: push reverses pull. It sets the output of the current vertex as op. If this vertex is pulled by others, the content of op will be returned.

Once \( F \) declared, together with an input graph \( G \), they encode a recursive dataflow graph structure, which maps to a subgraph of the implicit full dataflow graph of the model that may needs to be explicitly declared in traditional programming models. Via push and pull, Cavs allows users to connect any external static dataflow graph to a dynamic structure encoded by \((F, G)\), to express more complex model architectures, such as the LRCN [33] (i.e. connecting a CNN to an RNN), or an encoder-decoder LSTM network [136] (i.e. connecting two different recursive structures). With these four APIs, we present in Figure 6.7 an example user program how the \( N \)-ary child-sum Tree-LSTM [140] can be simply expressed by using them and other mathematical operators.

**Expressiveness**

With these four APIs, this new programming model can be seen as a middle ground between static and dynamic declaration. In the best case that the NN is fully recursive (e.g. most recurrent or recursive NNs), it can be represented by a single vertex function and an input graph. While in the worst case, that every sample has a unique input graph while every vertex in the graph has a unique way to interact with its neighboring vertices (i.e. the NN is dynamic but non-recursive nor recurrent), Cavs reduces to dynamic declaration that one has to define a vertex function for each vertex of each input graph. Fortunately, in general, dynamic NNs in this scenario are usually avoided because of the difficulties of design, programming and learning.
def $F()$:  
for $k$ in range($N$):  
S = gather($k$)  # gather states of child vertices  
c$_k$, h$_k$ = split($S$, 2)  # get hidden states c and h  
x = pull()  # pull the first external input $x$

# specify the computation  
h = $\sum_{k=0}^{N-1} h_k$  
i = sigmoid($W^{(i)}x + U^{(i)}h + b^{(i)}$)  
for $k$ in range($N$):  
f$_k$ = sigmoid($W^{(f)}x + U^{(f)}h + b^{(f)}$)  
o = sigmoid($W^{(o)}x + U^{(o)}h + b^{(o)}$)  
u = tanh($W^{(u)}x + U^{(u)}h + b^{(u)}$)  
c = $i \otimes u + \sum_{k=0}^{N-1} f_k \otimes c_k$  
h = $o \otimes$ tanh(c)  
scatter(concat([c, h], 1))  # scatter c, h to parents  
push(h)  # push to external connectors

Figure 6.7: The vertex function of an N-ary child-sum TreeLSTM [140] in Cavs. Within $F$, users declare a computational dataflow graph using symbolic operators. The defined $F$ will be evaluated on each vertex of $G$ following graph dependencies.

As a summary, we list the pros and cons of the three programming models – static declaration, dynamic declaration, and the vertex-centric programming model, in Table 6.1 for a side-by-side comparison.
<table>
<thead>
<tr>
<th>Programming Model</th>
<th>Frameworks</th>
<th>Expressiveness</th>
<th>Batching</th>
<th>Graph Construction Overhead</th>
<th>Graph Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>static declaration</td>
<td>Caffe, TensorFlow</td>
<td>×</td>
<td>√</td>
<td>low</td>
<td>beneficial</td>
</tr>
<tr>
<td>dynamic declaration (eager evaluation)</td>
<td>PyTorch, Chainer</td>
<td>√</td>
<td>×</td>
<td>N/A</td>
<td>unavailable</td>
</tr>
<tr>
<td>dynamic declaration (lazy evaluation)</td>
<td>DyNet</td>
<td>√</td>
<td>√</td>
<td>high</td>
<td>limited benefits</td>
</tr>
<tr>
<td>Fold</td>
<td>TensorFlow-Fold</td>
<td>√</td>
<td>√</td>
<td>high</td>
<td>unknown</td>
</tr>
<tr>
<td>Vertex-centric</td>
<td>Cavs</td>
<td>√</td>
<td>√</td>
<td>low</td>
<td>beneficial</td>
</tr>
</tbody>
</table>

Table 6.1: A side-by-side comparison of existing programming models for dynamic NNs, and their advantages and disadvantages.
6.5 Cavs: An Efficient Runtime System for Dynamic Neural Networks

Derived from the vertex-centric representation and interfaces, we next explain the design of the Cavs runtime system, by going through scheduling, memory management, and related system optimizations.

6.5.1 Scheduling

Once $F$ is defined and $G$ is obtained from I/O, Cavs will perform computation by scheduling the evaluation of $F$ over data samples $\{x_i\}_{i=1}^N$ and their input graphs $\{G_i\}_{i=1}^N$.

**Forward Pass**

For a sample $x_i$ with its input graph $G_i$, the scheduler starts the forward pass from the input vertices of $G_i$, and proceeds following the direction indicated by the edges in $G_i$. At each sub-step, the scheduler figures out the next activated vertex in $G_i$, and evaluates all expressions in $F$ at this vertex. It then marks this vertex as *evaluated*, and proceeds with the next activated vertex until reaching a terminal vertex (e.g. the loss function). A vertex of $G$ is activated if and only if all its dependent vertices have been evaluated.

**Backward Pass**

The backward pass is continued right after the forward. The scheduler first resets the status of all vertices as *not evaluated*, then scans the graph in a reverse direction, starting from the ending point of the forward pass. It evaluates $\partial F$ at each vertex until all vertices have been evaluated in the backward pass.

To train a NN to convergence, the above process has to be iterated on all samples $\{x_i\}_{i=1}^N$ and their input graphs $\{G_i\}_{i=1}^N$, for many epochs. We next describe our batched execution policy to speed the computation.

**Batching Policy in Scheduling**

Algorithm 5 illustrates the batched backpropagation process in Cavs. Particularly, given a data batch $\{x_k\}_{k=1}^K \subseteq \{x_i\}_{i=1}^N$ and associated graphs $\{G_k\}_{k=1}^K$, this policy groups multiple vertices and performs batched evaluation of $F$ in order to reduce kernel launches and exploit parallelism. Specifically, a forward pass over a batch $\{x_k\}_{k=1}^K$ are performed in multiple steps. At each step $t$, Cavs analyzes $\{G_k\}_{k=1}^K$ at runtime and determines a set $V_t$ that contains all activated vertices in graphs $\{G_k\}_{k=1}^K$. It then evaluates $F$ over these vertices by creating a *batched execution task*, with the task ID set to $t^4$. The task is executed by the Cavs execution engine (§6.6.3). Meanwhile, the scheduler records this task by pushing $V_t$ into a stack $S$. To perform backward pass, the

$^4$Whenever the context is clear, we use $V_t$ to denote both the set of vertices to be batched together, and the batched execution task itself.
Algorithm 5 Backpropagation with the batching policy.

1: function FORWARD(\{x_k\}_{k=1}^K, \{G_k\}_{k=1}^K, F)
2:   set task ID \(t \leftarrow 0\), task stack \(S \leftarrow \emptyset\).
3:   while NOT all vertices in \(\{G_k\}_{k=1}^K\) are evaluated do
4:     figure out all activated vertices in \(\{G_k\}_{k=1}^K\) as a set \(V_t\).
5:     push \(V_t\) into \(S\).
6:     evaluate \(F\) on \(V_t\): \(\text{GraphExecute}(V_t, F)\) (see §6.6.3).
7:     set the status of all vertices in \(V_t\) as evaluated.
8:     set \(t \leftarrow t + 1\).
9:   end while
10: return \(S\).
11: end function

12: function BACKWARD(\(S\), \(\{G_k\}_{k=1}^K\), \(\partial F\))
13:   set \(t\) as the size of \(S\).
14:   while \(S\) is not empty do
15:     pop the top element of \(S\) as \(V_t\).
16:     Evaluate \(\partial F\) on \(V_t\): \(\text{GraphExecute}(V_t, \partial F)\) (§6.5.4).
17:     set \(t \leftarrow t - 1\).
18:   end while
19: end function

scheduler pops out an element \(V_t\) from \(S\) at each step — the execution engine will evaluate the derivative function \(\partial F\) over vertices in \(V_t\), until all vertices of \(\{G_k\}_{k=1}^K\) are evaluated.

We note the batching policy is similar to the dynamic batching in Fold [94] and DyNet [109]. However, Cavs determines how to batch fully dynamically during runtime using simple breadth-first search with negligible cost (instead of analyzing full dataflow graphs before every iteration of the execution). Since batched computation requires the inputs to an expression over multiple vertices to be placed on a continuous memory buffer, we develop a new memory management support for it.

6.5.2 Memory Management

In static declaration [3, 109], a symbol in the user program usually corresponds to a fixed-sized tensor object with a batch size dimension. While in Cavs, each batching task \(V_t\) is determined at runtime. For the batched computation to be efficient, Cavs must guarantee for each batching task, the inputs to each expression of \(F\) over a group of runtime-determined vertices coalescing in memory.

Cavs proposes a novel data structure dynamic tensor to address this challenge (Figure 6.8). A dynamic tensor is a wrapper of a multi-dimensional array [3, 147]. It contains four attributes: shape, \(bs\), a pointer \(p\) to a chunk of memory, and offset. shape is an array of integers representing the specific shape of the tensor excluding the batch dimension. It can be inferred from the user program and set before execution. The batch size \(bs\) is dynamically set by the scheduler at runtime at the beginning of a batching task. To access a dynamic tensor, one moves \(p\) forward
with the value of `offset`, and reads/writes number of elements equal to \(bs \cdot \prod_i \text{shape}[i]\). Therefore, `bs` together with `offset` provide a view of the tensor, and the state of the tensor will vary based on their values. Given a vertex function \(\mathcal{F}\), Cavs creates dynamic tensors \(\{\alpha_n\}_{n=1}^N\) for each non-parameter symbol \(s_n(n = 1, \ldots, N)\) in \(\mathcal{F}\), and also \(\{\nabla \alpha_n\}_{n=1}^N\) as their gradients, while it creates static tensors for model parameters.

Figure 6.9 illustrates how the memory is assigned during the forward pass by manipulating dynamic tensors. In particular, in a training iteration, for a batching task \(V_t\), the scheduler sets `bs` of all \(\{\alpha_n\}_{n=1}^N\) to \(M_t = |V_t|\) (the number of vertices in \(V_t\)). The execution engine then performs batched evaluation of each expression in \(\mathcal{F}\). For an expression \(s_t = \text{op}(s_r)\), Cavs first accesses \(\alpha_r\) (the dynamic tensor of the RHS symbol \(s_r\)) – it offsets \(\alpha_r.p\) by \(\alpha_r.offset\), and reads a block of \(M_t \prod_i \alpha_r.shape[i]\) elements, and presents it as a tensor with batch size \(M_t\) and other dimensions as \(\alpha_r.shape\). It then applies batched computational kernels of the operator \(\text{op}\) over this memory block, and writes the results to \(\alpha_l\) (the dynamic tensor of the LHS symbol \(s_l\)) on the continuous block in between \([\alpha_l.p + \alpha_l.offset, \alpha_l.p + \alpha_l.offset + M_t \prod_i \alpha_l.shape[i]]\). Upon the completion of \(V_t\), the scheduler increases `offset` of all \(\{\alpha_n\}_{n=1}^N\) by \(M_t \prod_i \alpha_n.shape[i]\), respectively. It then starts the next task \(V_{t+1}\). Hence, intermediate results generated in each batching task at forward pass are stored continuously in the dynamic tensors, and their offsets are recorded.

At the entrance of \(\mathcal{F}\), the vertices \(\{v_m\}_{m=1}^{M_t}\) in \(V_t\) need to interact with its dependent vertices in previous \(V_{t-1}\) to gather their outputs as inputs (L3 of Figure 6.7), or pull inputs from the external (L5 of Figure 6.7). Cavs maintains memory buffers to enable this (Figure 6.9). It records the offsets of the dynamic tensors for each \(v_m \in V_t\), and therefore during the execution of `gather` operator, the memory slices of specific children can be indexed. As shown in Figure 6.9, `gather` and `scatter` share the same temporary buffer for memory re-organization, but `push` and `pull` operate on external memory buffers.

Algorithm 6 summarizes the memory management during forward and backward pass. The backward execution follows an exactly reverse order of the forward pass (§6.5.1), which we skip in the text. With this strategy, Cavs guarantees memory continuity for any batched computation of \(\mathcal{F}\) and \(\partial \mathcal{F}\). Compared to dynamic batching in DyNet, Cavs performs memory movement only at the entrance and exit of \(\mathcal{F}\), instead of for each expression (operator). We empirically find this significantly reduces overhead of memory operations (§6.6.4).

5Note that the user-defined expressions can be arbitrary, e.g. with more than one argument or return values
Algorithm 6 Memory management at forward and backward pass.

1: function FORWARD($V_t \{i=1\}^{T}$, $\{\alpha_n\}_{n=1}^{N}$, $\mathcal{F}$)
2:     for $t = 1 \rightarrow T$ do
3:         for $n = 1 \rightarrow N$ do $\alpha_n.bs \leftarrow M_t$ end for
4:         for each expression like $s_i = \text{op}(s_r)$ in $\mathcal{F}$ do
5:             if $\text{op} \in \{\text{gather, pull}\}$ then
6:                 $C \leftarrow \prod_i \alpha_i.shape[i], q \leftarrow \alpha_i.p + \alpha_i.offset.$
7:                 for $v_m \in V_i(m = 1 \rightarrow M_t)$ do
8:                     src $\leftarrow \text{IndexBuffer}(\text{op}, m), \text{dest} \leftarrow q + (m - 1)C.$
9:                     memcpy(\text{dest, src}, C).
10:             end for
11:         else if $\text{op} \in \{\text{scatter, push}\}$ then
12:             $C \leftarrow \prod_i \alpha_r.shape[i], q \leftarrow \alpha_r.p + \alpha_r.offset.$
13:             for $v_m \in V_i(m = 1 \rightarrow M_t)$ do
14:                 dest $\leftarrow \text{IndexBuffer}(\text{op}, m), \text{src} \leftarrow q + (m - 1)C.$
15:                 memcpy(\text{dest, src}, C).
16:             end for
17:         else
18:             perform batched computation: $\alpha_l = \text{op}_\text{kernel}(\alpha_r)$.
19:             end if
20:         end for
21:     for $n = 1 \rightarrow N$ do $\alpha_n.offset+ = M_t \prod_i \alpha_n.shape[i]$ end for
22: end for
23: end function
24: function BACKWARD($V_t \{i=1\}^{T}$, $\{\alpha_n\}_{n=1}^{N}$, $\{\nabla \alpha_n\}_{n=1}^{N}$, $\partial\mathcal{F}$)
25:     for $n = 1 \rightarrow N$ do $\nabla \alpha_n.offset \leftarrow \alpha_n.offset$ end for
26:     for $t = T \rightarrow 1$ do
27:         for $n = 1 \rightarrow N$ do $\alpha_n.bs \leftarrow M_t, \nabla \alpha_n.bs \leftarrow M_t$ end for
28:         for an expression $\nabla s_r = \text{grad}_{\text{op}}(\nabla s_l, s_l, s_r)$ in $\partial\mathcal{F}$ do
29:             for $\alpha \in \{\alpha_l, \alpha_r, \nabla \alpha_l, \nabla \alpha_r\}$ do
30:                 $\alpha.offset \leftarrow \alpha.offset - M_t \prod_i \alpha.shape[i]$
31:             end for
32:         if $\text{grad}_{\text{op}} \in \{\text{gather, pull}\}$ then
33:             $C \leftarrow \prod_i \nabla \alpha_r.shape[i], q \leftarrow \nabla \alpha_r.p + \nabla \alpha_r.offset.$
34:             for $v_m \in V_i(m = 1 \rightarrow M_t)$ do
35:                 src $\leftarrow \text{IndexBuffer}(\text{op}, t_m), \text{dest} \leftarrow q + (m - 1)C.$
36:                 memcpy(\text{dest, src}, C).
37:             end for
38:         else if $\text{grad}_{\text{op}} \in \{\text{scatter, push}\}$ then
39:             $C \leftarrow \prod_i \nabla \alpha_l.shape[i], q \leftarrow \nabla \alpha_l.p + \nabla \alpha_l.offset.$
40:             for $v_m \in V_i(m = 1 \rightarrow M_t)$ do
41:                 dest $\leftarrow \text{IndexBuffer}(\text{op}, t_m), \text{src} \leftarrow q + (m - 1)C.$
42:                 memcpy(\text{dest, src}, C).
43:             end for
44:         else
45:             $\nabla \alpha_l, \alpha_l, \alpha_r$, if necessary, perform batched computation following $\nabla \alpha_r = \text{grad}_{\text{op}}(\nabla \alpha_l, \alpha_l, \alpha_r), \text{write the (batch) results continuously to } \nabla \alpha_r$.
46:         end if
47:     end for
48: end for
49: end function
def \( F() \):
    \( a_0 = \) pull()
    \( a_1 = \) gather(0)
    \( a_2 = \) gather(1)
    \( a_3 = \) matmul\( (a_0, u) \)
    \( + \) matmul\( (a_1, v) \)
    \( + \) matmul\( (a_2, w) \)
    scatter\( (0, a_3) \)
    push\( (a_3) \)

Figure 6.9: The memory management at the forward pass of \( F \) (top-left) over two input trees (bottom-left). Cavs first analyzes \( F \) and inputs – it creates four dynamic tensors \( \{a_n\}_{n=0}^3 \), and figures out there will be four batch tasks (dash-lined boxes). Starting from the first task (orange vertices \( \{0, 1, 2, 5, 6, 7, 8, 9\} \)), Cavs performs batched evaluation of each expression in \( F \). For example, for the pull expression \( a_0 = \) pull\( () \), it indexes the content of \( a_0 \) on all vertices from the pull buffer using their IDs, and copies them to \( a_0 \) continuously; for scatter and push expressions, it scatters a copy of the output \( (a_3) \) to the gather buffer, and pushes them to the push buffer, respectively. Cavs then proceeds to the next batching task (blue vertices). At this task, Cavs evaluates each expression of \( F \) once again for vertices \( \{3, 10, 11\} \). (e.g. for a pull expression \( a_0 = \) pull\( () \), it pulls the content of \( a_0 \) from pull buffer again; for a gather expression \( a_2 = \) gather\( (1) \) at vertex 3, it gathers the output of the second child of 3, which is 1); it writes results continuously at the end of each dynamic tensor. It proceeds until all batching tasks are finished.
6.5.3 Auto-differentiation

Cavs by nature supports auto-differentiation. Given a vertex function $F$ it derives $\partial F$ following the auto-differentiation rules: for each math expression such as $s_i = \text{op}(s_r)$ in $F$, Cavs generates a corresponded backward expression in $\partial F$ as $\nabla s_r = \text{grad}_{\text{op}}(\nabla s_l, s_l, s_r)$. For the four proposed operators, with the memory management strategy described above, we note scatter is the backward operator of gather in the sense that if $\text{gather}$ collects inputs from $\text{gatherBuffer}$ previously written by $\text{scatter}$ at the forward pass, a $\text{scatter}$ needs to be performed to write the gradients to the $\text{gatherBuffer}$ for its dependent vertices to $\text{gather}$ at the backward pass. Hence, for an expression like $s_i = \text{gather}(\text{child}_\text{idx})$ in $F$, Cavs will generate a backward expression $\text{scatter}(\nabla s_l)$ in $\partial F$. Similarly, the gradient operator of $\text{scatter}$ is $\text{gather}$. The same auto-differentiation rule applies for $\text{push}$ and $\text{pull}$ as well.

6.5.4 Optimizing Execution Engine

Since Cavs separates out a static dataflow graph encoded by $F$, we can replace the original $F$ with an optimized one that runs more efficiently, as long as maintaining correctness. We next described our optimization strategies.

Lazy Batching and Streaming

In addition to batched execution of $F$, the lazy batching and streaming explore potential parallelisms for a certain group of finer-grained operators in $F$ or $\partial F$ called lazy and eager operators. Note that streaming is a borrowed terminology from CUDA which means executing different commands concurrently with respect to each other on different GPU streams. As Cavs’ optimizations are agnostic to the low-level hardware, we use streaming interchangeably with multi-threading if the underlying computing hardware is CPU.

**Definition 6.5.1** An operator in $F$ ($\partial F$) is a lazy operator if at the forward (backward) pass, for $\forall v \in \mathcal{G}, \forall \mathcal{G} \in \{\mathcal{G}_k\}_{k=1}^K$, the evaluation of $F$ ($\partial F$) at any parent (dependent) vertex of $v$ does not
rely on the evaluation of \( F \) at \( v \). It is an eager operator if the evaluation at \( v \) does not rely on the evaluation of \( F \) (\( \partial F \)) at any dependents (parents) of \( v \).

**Proposition 6.5.2** Denote \( D_F (D_{\partial F}) \) as the dataflow graph encoded by \( F (\partial F) \), and \( g, s \in D_F \) (\( D_{\partial F} \)) as nodes of gather and scatter operator, respectively. A node that has \( g \) as its dependent and is not on any path from \( g \) to \( s \) is a lazy operator. A node that has \( s \) as its ancestor and is not on any path from \( g \) to \( s \) is an eager operator.

Figure 6.10 illustrates a forward dataflow graph of the vertex function of Tree-LSTM, with eager and lazy operators colored. A property of them is that their evaluation is not fully subject to the dependency reflected by the input graph \( G \). For instance, the pull operator in Figure 6.10 is eager and can be executed in prior – even before \( F \) has been evaluated at the vertices that gather tries to interact with; the push operator is lazy, so we can defer its execution without impacting the evaluation of \( F \) at parent vertices. Similarly, in \( \partial F \), the gradient derivation for model parameters are mostly lazy – their execution can be deferred as long as the gradients of hidden states are derived and propagated in time. Cavs leverages this property and proposes the lazy batching strategy. It defers the execution of all lazy operators in \( F \) and \( \partial F \) until all batching tasks \( \{V_t\}_{t=1}^T \) has finished. It then performs a batched execution of these lazy operators over all vertices of \( \{G_k\}_{k=1}^K \). These operators includes, but is not limited to, the push operator that is doing memory copy, and operators for computing gradients of model parameters. Lazy batching helps exploit more parallelism and significantly reduces kernel launches. Empirically lazy batching brings 20% overall improvement (§6.6.4).

To leverage the exhibited parallelization opportunity between eager operators and the operators on the path from gather to scatter (Figure 6.10), Cavs proposes a streaming strategy that pipelines the execution of these two groups of operators. It allocates two streams, and puts the eager operators on one stream, and the rest (excluding lazy operators) on the other. Hence, independent operators in two streams run in parallel, while for those operators that depend on an eager operator, this dependency is respected by synchronization barriers.

**Automatic Kernel Fusion**

Given \( F \), before execution, Cavs will run a fusion detector \([52]\) to scan its corresponded dataflow graph and report all fuse-able subgraphs therein, i.e. all nodes in a fuse-able subgraph can be fused as a single operator that behaves equivalently but takes less execution time (e.g. with fewer kernel launches and I/O, or faster computation). Currently, we only detect groups of directly linked elementwise operators, such as \( +, \text{sigmoid} \), as shown in Figure 6.10, and we use a simple union-find algorithm to detect the largest possible fuse-able subgraphs. Given a fuse-able subgraph, Cavs adopts de facto automatic code generation techniques \([27, 111, 117, 119]\) to generate lower-level kernel implementations. Replacing the original fuse-able subgraphs with fused operators during execution is beneficial in many aspects: (1) it reduces the number of kernel launches; (2) on some devices such as GPUs, kernel fusion transform device memory access into faster device registers access. We empirically report another 20% improvement with automatic kernel fusion (§6.6.4).
6.5.5 Implementation

We implement Cavs as a pluggable C++ library that can be integrated with existing DL frameworks to provide or enhance their support for dynamic NNs. We next briefly discuss implementation details. For clarity, we assume the host framework is composed of three major layers (which is the case for most popular frameworks [3, 7, 109]): (1) a frontend that provides device-agnostic symbolic programming interface; (2) an intermediate layer that implements the core execution logic; (3) a backend with device-specific kernels for all provided operators.

Frontend

Cavs provides a base class GraphSupport in addition to conventional operators and the four proposed APIs. Users are required to instantiate it by providing a symbolic vertex function $F$ – therefore an instantiation of GraphSupport represents a single dynamic structure. To construct more complex structures (e.g. encoder-decoder LSTM [136], LRCN [33]), users employ push and pull to connect this dynamic structure to external structures.

Intermediate Layer

At the intermediate layer, Cavs will create a unique scope [3], and generates a small dataflow graph for each instantiation of GraphSupport, connecting them appropriately with other parts of the model according to user programs. Cavs implements its core runtime logic at this layer, i.e. the scheduler, the memory management, and the graph execution engine, etc. During execution, the execution engine first analyzes the received dataflow graphs and incorporates described optimization in §6.5.4. The scheduler then instructs the system to read training samples and their associates graphs (e.g., adjacency matrices). It starts training by submitting batching tasks to the execution engine and assigning memory accordingly.

Backend

Following common practice [3, 39, 109], Cavs puts device-specific kernel implementations for each supported operator at this layer. Each operator implementation is a function that takes as inputs static tensors and produces static tensors as outputs – therefore the higher-layer logic, i.e. how the computation is scheduled or how the memory is assigned are invincible to this layer. Cavs will reuse the native operator implementations from the host framework, while it provides optimized implementations for the four proposed primitives (gather, scatter, pull, push). Specifically, gather and pull index different slices of a tensor and puts them together continuously on memory; scatter and push by contrast splits a tensor along its batch dimension, and copy different slices to different places. Cavs implements a customized memcpy kernel for there four operators so that copying multiple slices from (or to) different places is performed within one kernel, to further reduce kernel launches.
6.6 Evaluation

In this section, we evaluate Cavs on multiple NNs and datasets, obtaining the following major findings:

- Cavs has little overhead: on static NNs, Cavs demonstrates equal performance on training and inference with other systems; On several NNs with notably difficult-to-batch structures, Cavs outperforms all existing frameworks by a large margin.

- We confirm the graph construction overhead is substantial in both Fold [94] and dynamic declaration [109], while Cavs bypasses it by loading input graphs through I/O.

- We verify the effectiveness of our proposed design and optimization via ablation studies, and discuss Cavs’ advantages over other DL systems for dynamic dataflow graphs.

6.6.1 Experiment Setup

Environment

We perform all experiments in this paper on a single machine with an NVIDIA Titan X (GM200) GPU, a 16-core CPU, and CUDA v8.0 and cuDNN v6 installed. As modern DL models are mostly trained using GPUs, we focus our evaluation on GPUs, but note Cavs’ design and implementation do not rely on a specific type of device. We mainly compare Cavs to TensorFlow v1.2 [3] with XLA [44] and its variant Fold [94], PyTorch v0.3.0 [38], and DyNet v2.0 [109] with autobatching [110], as they have reported better performance than other frameworks [14, 144] on dynamic NNs. We focus on metrics for system performance, e.g. time to scan one epoch of data. Cavs produces exactly the same numerical results with other frameworks, hence the same per-epoch convergence.

Models and Dataset

We experiment on the following models with increasing difficulty to batch:

- **Fixed-LSTM** language model (LM): a static sequence LSTM with fixed steps for language modeling [135, 136, 166]. We train it using the PTB dataset [143] that contains over 10K different words. We set the number of steps as 64, i.e. at each iteration of training, the model takes a 64-word sentence from the training corpus, and predicts the next word of each word therein. Obviously, the computation can be by nature batched easily, as each sentence has exactly the same size.
• **Var-LSTM** LM: that accepts variable-length inputs. At each iteration the model takes a batch of natural sentences with different length from PTB, and predicts the next words;

• **Tree-FC**: the benchmarking model used in [94] with a single fully-connected layer as its cell function. Following the same setting in [94], we train it over synthetic samples generated by their code [142] – each sample is associated with a complete binary tree with 256 leaves (therefore 511 vertices per graph);

• **Tree-LSTM**: a family of dynamic NNs widely adopted for text analysis [90, 146]. We implement the binary child-sum Tree-LSTM in [140], and train it as a sentiment classifier using Stanford sentiment treebank (SST) dataset [132]. The dataset contains 8544 training sentences, each associated with a human annotated grammar tree, and the longest one has 54 words.

### 6.6.2 Overall Performance

#### Fixed-LSTM

We first verify the viability of our design on the easiest-to-batch case: Fixed-LSTM language model. We compare Cavs to three strong baselines:

• **CuDNN** [18]: a CuDNN-based fixed-step sequence LSTM, which is highly optimized by NVIDIA using handcrafted kernels and stands as the best performed implementation on NVIDIA GPUs;

• **TF**: the official implementation of Fixed-LSTM LM in TensorFlow repository [141] based on static declaration;

• **DyNet**: we implement a 64-step LSTM in DyNet based on dynamic declaration – we declare a dataflow graph per sample, and train with the autobatching [110] enabled;

• **Cavs** with batching policy, and all input samples have a same input graph – a 64-node chain.

We train the model to converge, and report the average time per epoch in Figure 6.11(a)(e), where in (a) we fix the hidden size $h$ of the LSTM unit as 512 and vary the batch size $bs$, and in (e) we fix $bs = 64$ and vary $h$. Empirically, CuDNN performs best in all cases, but note it is highly inflexible. Cavs performs slightly better than TF in various settings, verifying that our system has little overhead handling fully static graphs, though it is specialized for dynamic ones. We also conclude from Figure 6.11 that batching is essential for GPU-based DL: $bs = 128$ is nearly one order of magnitude faster than $bs = 1$ regardless of used frameworks. For Cavs, the batching policy is $1.7x$, $3.8x$, $7.0x$, $12x$, $15x$, $25x$, $36x$ faster than non-batched at $bs = 2, 4, 8, 16, 32, 64, 128$, respectively.

#### Var-LSTM

Next, we experiment with Var-LSTM, the most commonly used RNN for variable-length sequences. We compare the following three implementations (CuDNN-based LSTM cannot handle
variable-length inputs):

- **TF**: an official TensorFlow implementation based on the dynamic unroll approach described in §6.7;
- **DyNet**: an official implementation from DyNet benchmark repository based on dynamic declaration [36];
- **Cavs**: where each input sentence is associated with a chain graph that has number of vertices equal to the number of words.

We vary $h$ and $bs$, and report the results in Figure 6.11(b)(f), respectively. Although all three systems perform batched computation in different ways, **Cavs** is consistently 2-3 times faster than **TF**, and outperforms **DyNet** by a large margin. Compared to **TF**, **Cavs** saves computational resources. **TF** dynamically unrolls the LSTM unit according to the longest sentence in the current batch, but it cannot prevent unnecessary computation for those sentences that are shorter than the longest one.

**Tree-FC**

We then turn to **Tree-FC**, a dynamic model for benchmarking. Since vanilla TensorFlow is unable to batch its computation, we compare **Cavs** to (1) **DyNet** and (2) **Fold**, a specialized library built upon TensorFlow for dynamic NNs, with a depth-based dynamic batching strategy. To enable the batching, it however needs to preprocess the input graphs, translate them into intermediate representations and pass them to lower-level TensorFlow control flow engine for execution. We report the results in Figure 6.11(c)(g) with varying $bs$ and $h$, respectively. For all systems, we allocate a single CPU thread for graph preprocessing or construction. **Cavs** shows at least an order of magnitude speedups than **Fold** and **DyNet** at $h \leq 512$. Because the size of the synthetic trees is large, one major advantage of **Cavs** over them is the alleviation of graph preprocessing/construction overhead. With a single CPU thread, **Fold** takes even more time on graph preprocessing than computation (§6.6.4).

**Tree-LSTM**

Finally, we compare three frameworks on **Tree-LSTM** in Figure 6.11(d)(h): **Cavs** is 8-10x faster than **Fold**, and consistently outperforms **DyNet**. One difference in this experiment is that we allocate as many CPU threads as possible (32 on our machine) to accelerate graph preprocessing for **Fold**, otherwise it will take much longer time. Further, we note **DyNet** performs much better here than on **Tree-FC**, as the size of the input graphs in SST (maximally 54 leaves) is much smaller than the synthetic ones (256 leaves each) in **Tree-FC** experiments. We observe **DyNet** needs more time on graph construction for large input graphs, and **DyNet**’s dynamic batching is less effective on larger input graphs, as it has to perform frequent memory checks to support its dynamic batching, which we will discuss in §6.6.4. We also compare **Cavs** with **PyTorch** – its per-epoch time on **Tree-LSTM** is 542s, 290x slower than **Cavs** when the batch size is 256. Compared to other systems, **PyTorch** cannot batch the execution of dynamic NNs.
Figure 6.11: Comparing five systems on the averaged time to finish one epoch of training on four models: Fixed-LSTM, Var-LSTM, Tree-FC and Tree-LSTM. In (a)-(d) we fix the hidden size $h$ and vary the batch size $bs$, while in (e)-(h) we fix $bs$ and vary $h$. 
Figure 6.12: The averaged graph construction overhead per epoch when training (a) Tree-FC with different size of input graphs (b) Tree-LSTM with different batch size. The curves show absolute time in second (left y-axis), and the bar graphs show its percentage of the overall time (right y-axis).

### 6.6.3 Graph Construction

In this section, we investigate the graph construction overhead in Fold and DyNet. To batch computation of different graphs, Fold analyzes the input graphs to recognize batch-able dynamic operations, then translates them into intermediate instructions, with which, TensorFlow generates appropriate control flow graphs for evaluation – we will treat the overhead caused in both steps as Fold’s graph construction overhead. DyNet, as a typical dynamic declaration framework, has to construct as many dataflow graphs as the number of samples. Though DyNet has optimized its graph construction to make it lightweight, the overhead still grows with the training set and the size of input graphs. By contrast, Cavs takes constant time to construct a small dataflow graph encoded by $F$, then reads input graphs through I/O. To quantify the overhead, we separate the graph construction from computation, and visualize in Figure 6.12(a) how it changes with the average number of leaves (graph size) of input graphs on training Tree-FC, with fixed $bs = 64, h = 512$. We compare (1) Cavs (2) Fold-1 which is Fold with one graph processing thread and (3) DyNet. We plot for one epoch, both the (averaged) absolute time for graph construction and it percentage of the overall time. Clearly we find that all three systems take increasingly more time when the size of the input graphs grows, but Cavs, which loads graphs through I/O, causes the least overhead at all settings. In terms of the relative time, Fold unfortunately wastes 50% at 32 leaves, and 80% when the tree has 1024 leaves, while DyNet and Cavs take only 10% and 20%, respectively.

We also wonder how the overhead is related with batch size when there is fixed computational workload. We report in Figure 6.12(b) the same metrics when training Tree-LSTM with varying $bs$. We add another baseline Fold-32 with 32 threads for Fold’s graph preprocessing. As Fold-1 takes much longer time than others, we report its time at $bs = 1, 16, 32, 64, 128, 256$ here (instead of showing in Figure 6.12): 1.1, 7.14, 31.35, 40.1, 46.13, 48.77. Except $bs =
Table 6.2: The averaged computation time (Cavs/Fold/DyNet) and the speedup (Cavs vs Fold/DyNet) for training one epoch on Tree-FC with varying size of the input trees (left part), and on Tree-LSTM with varying batch size (right part).

<table>
<thead>
<tr>
<th># leaves</th>
<th>time (s)</th>
<th>Speedup</th>
<th>bs</th>
<th>time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0.6 / 3.1 / 4.1</td>
<td>5.4 / 7.1</td>
<td>1</td>
<td>76 / 550 / 62</td>
<td>7.2 / 0.8</td>
</tr>
<tr>
<td>64</td>
<td>1.1 / 3.9 / 8.0</td>
<td>3.7 / 7.5</td>
<td>16</td>
<td>9.8 / 69 / 12</td>
<td>7.0 / 1.2</td>
</tr>
<tr>
<td>128</td>
<td>2 / 6.2 / 16</td>
<td>3.0 / 7.9</td>
<td>32</td>
<td>6.2 / 43 / 9.9</td>
<td>7.0 / 1.6</td>
</tr>
<tr>
<td>256</td>
<td>4 / 10.6 / 33.7</td>
<td>2.7 / 8.7</td>
<td>64</td>
<td>4.1 / 29 / 7.4</td>
<td>7.2 / 1.8</td>
</tr>
<tr>
<td>512</td>
<td>8 / 18.5 / 70.6</td>
<td>2.3 / 8.7</td>
<td>128</td>
<td>2.9 / 20.5 / 5.9</td>
<td>7.1 / 2.0</td>
</tr>
<tr>
<td>1024</td>
<td>16 / 32 / 153</td>
<td>2.1 / 9.7</td>
<td>256</td>
<td>2.3 / 15.8 / 5.4</td>
<td>7.0 / 2.4</td>
</tr>
</tbody>
</table>

In all three systems (except Fold-1) take almost constant time for graph construction in one epoch, regardless of $bs$, while Fold-32 and DyNet take similar time, but Cavs takes 20x less. Nevertheless, at the percentage scale, increasing $bs$ makes this overhead more prominent, because larger batch size yields improved computational efficiency, therefore less time to finish one epoch. This, from one perspective, reflects that the graph construction is a main obstacle that grows with the number of training samples and prevents the efficient training of dynamic NNs in existing frameworks, while Cavs successfully overcomes this barrier.

Apart from the graph construction we report in Table 6.2 the computation-only time. Cavs shows maximally 5.4x/9.7x and 7.2x/2.4x speedups over Fold/DyNet on Tree-FC and Tree-LSTM, respectively. The advantages stem from two main sources: an optimized graph execution engine, and a better-suited memory management strategy, which we investigate next.

### 6.6.4 Graph Optimizations

#### Graph Execution Engine

To reveal how much each optimization in §6.5.4 contributes to the final performance, we disable lazy batching, fusion and streaming in Cavs and set this configuration as a baseline (speedup = 1). We then turn on one optimization at a time and record how much speedup it brings. We train Fixed-LSTM and Tree-LSTM, and report the averaged speedups one computation-only time in one epoch over the baseline configuration in Figure 6.13, with $bs = 64$ but varying $h$.

Lazy batching and fusion consistently deliver nontrivial improvement – lazy batching is more beneficial with a larger $h$ while fusion is more effective at smaller $h$, which are expected: lazy batching mainly parallelizes matrix-wise operations (e.g. matmul) commonly with $O(h^2)$ or higher complexity, while fusion mostly works on elementwise operations with $O(h)$ complexity [51].

Streaming, compared to the other strategies, is less effective on Tree-LSTM than on Fixed-LSTM, as we have found the depth of the input trees in SST exhibit high variance, i.e. some trees are much deeper than others. In this case, many batching tasks only have one vertex to be evaluated. The computation is highly fragmented and the efficiency is bounded by kernel launching latency. Lazy batching and fusion still help as they both reduce kernel launches (§6.5.4). Streaming, which tries to pipeline multiple kernels, can hardly yield obvious improvement.
Table 6.3: Breakdowns of average time per epoch on memory-related operations and computation, comparing Cavs to DyNet on training and inference of Tree-LSTM with varying $bs$.

<table>
<thead>
<tr>
<th>$bs$</th>
<th>Memory operations (s) ($Cavs / DyNet$)</th>
<th>Computation (s) ($Cavs / DyNet$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Train</td>
<td>Inference</td>
</tr>
<tr>
<td>16</td>
<td>1.14 / 1.33</td>
<td>0.6 / 1.33</td>
</tr>
<tr>
<td>32</td>
<td>0.67 / 0.87</td>
<td>0.35 / 0.87</td>
</tr>
<tr>
<td>64</td>
<td>0.39 / 0.6</td>
<td>0.21 / 0.6</td>
</tr>
<tr>
<td>128</td>
<td>0.25 / 0.44</td>
<td>0.13 / 0.44</td>
</tr>
<tr>
<td>256</td>
<td>0.17 / 0.44</td>
<td>0.09 / 0.44</td>
</tr>
</tbody>
</table>

Memory Management

Cavs’ performance advantage also credits to its memory management that reduces memory movements while guarantees continuity. Quantitatively, it is difficult to compare Cavs to Fold, as Fold relies on TensorFlow where memory management is highly coupled with other system aspects. Qualitatively, we find Cavs requires less memory movement (e.g. memcpy) during dynamic batching. Built upon the `tf.while` operator, whenever Fold performs depth-based batching at depth $d$, it has to move all the contents of nodes in the dataflow graphs at depth $d - 1$ to a desired location, as the control flow does not support cross-depth memory indexing. This results in redundant memcpy, especially when the graphs are highly skewed. By contrast, Cavs only copies contents that are necessary to the batching task. DyNet has a specialized memory management strategy for dynamic NNs. Compared to Cavs, it however suffers substantial overhead caused by repeated checks of the memory continuity – whenever DyNet wants to batch operators with same signatures, it checks whether their inputs are continuous on memory [110]. The checking overhead increases with $bs$ and is more prominent on GPUs. Thanks to the sim-
plicity of both systems, we are able to profile the memory-related overhead during both training and inference, and separate it from computation. We compare them on TreeLSTM, and report the breakdown time per epoch in Table 6.3 under different $bs$. We observe the improvement is significant (2x - 3x) at larger $bs$, especially during inference where DyNet has its continuity checks concentrated.

### 6.7 Additional Related Work

**Graph Execution Optimization**

Optimizing the execution of DL dataflow graphs comes in mainly two ways: better operator implementations or optimizing the execution of (sub-)graphs. As Cavs is implemented as a plug-in to enhance existing frameworks, it benefits from any improved implementations of specific operators (e.g. cuDNN) [18, 47, 50, 65]. In addition, Cavs has optimized implementations for its proposed four primitives (gather/scatter/pull/push). At the graph level, a variety of well-developed techniques from other areas, such as kernel fusion, common subexpression elimination, and constant folding, have been adapted and applied on speeding the computation of DL dataflow graphs [3, 14, 39, 44]. They are usually incorporated after the graph declaration, but before the execution, so that the actual computation is conducted on an optimized graph other than the original one. However, these graph optimizations are less beneficial in dynamic declaration, in which the graph changes with the sample, and needs to be re-processed and re-optimized every iteration, and may cause substantial overhead. On the contrary, Cavs separates the static vertex function from the dynamic-varying input graph, so it benefits from most of the aforementioned optimizations, as we have shown in §6.6.4. We draw insights from these strategies and reflect them in Cavs’ execution engine. We further propose lazy batching and streaming to exploit more parallelism exposed by our programming model.

**Vertex-centric Models**

The vertex-centric programming model has been extensively developed in the area of graph computing [12, 42, 96, 134]. Cavs draws insights from the GAS model [42], but faces totally different challenges in system and interface design, such as expressiveness, scheduling for batched execution of different graphs, guaranteeing the memory continuity, etc., as we have discussed in §6.4.2.

**Dynamic Neural Network Frameworks**

Tensorflow Fold [94] also performs dynamic batching for dynamic dataflow graphs. Fold turns dynamic dataflow graphs into a static control flow graph to enable batched execution, but introduces a complicated functional programming-like languages and a large graph preprocessing overhead. There are also some “imperative” frameworks, such as PyTorch [38] and Chainer [144] that allow users to construct dynamic NNs but performs instant evaluation of each user expression. As model construction and execution are coupled, they are usually difficult to perform...
dynamic batching or graph optimization. Overall, they are still far from efficient when handling dynamic NNs.
Chapter 7

Composable Distributed Parallelization

The largest ML models nowadays may involve 100s of millions to billions of parameters [82, 129] if they are dense models, or even trillions [32, 118, 164] for sparse models. To cope with the computational cost of training such large models on large datasets, software systems have been created to parallelize training algorithms over multiple CPUs or GPUs (multi-device parallelism) [81, 114, 168], and multiple computing devices over a network (distributed parallelism) [3, 23, 167, 169]. These software systems exploit ML-theoretic properties to achieve speedups scaling with the number of devices. Ideally, such parallel ML systems should democratize training of large models on large datasets, by reducing training times to allow quicker research iteration. In practice, the majority of papers at ML conferences and journals (such as NeurIPS, ICML, KDD, AAAI, JMLR, etc.) report experiments taken on single workstations.

What are the opportunities to increase the adoption of parallel ML systems, putting aside the cost of hardware\(^1\)? We believe these opportunities are:

1. **ML systems either require high expertise, or are narrowly-specialized**: they either require low-level programming to specify how to partition distributed operations [3, 104, 114], or are effective only on a limited set of models [13, 126];

2. **Multiple parallel ML strategies\(^2\) can be composed to achieve near-ideal performance, but we need new interfaces and systems for composition**: our research has shown that different parallel ML strategies can be applied to different parts of an ML model, in order to achieve near-linear (i.e. ideal) increases in throughput [169]. Composing multiple strategies is technically demanding, and new systems and programming interfaces are needed to increase accessibility to ML practitioners;

3. **ML systems should be composable, to facilitate rapid adaptation to unseen ML models**: ML systems are usually monolithic software projects, and it is practically impossible to apply multiple parallel ML strategies to the same ML model. Decomposing these ML

\(^1\)We note that the 2020 cost of 4 workstations, each with 4 top-end consumer GPUs, is under USD $40,000. With the right parallel ML system, this setup should enable practitioners to train models over 10 times faster than a single-GPU workstation.

\(^2\)In this chapter, we use the phrases “parallel ML strategy” and “ML parallelism” interchangeably. We use the term “aspects” to refer to orthogonal spaces of design choices in a parallel ML program.
systems into composable units will allow fast experimentation on newly developed ML models, which will likely require new parallel ML strategies.

4. **Distributed-parallel ML strategies are hard to choose**: different distributed strategies (such as parameter server, all-reduce, peer-to-peer, quantization and prefix coding, etc.) exhibit sharp differences in performance when applied to different ML model building blocks (Bayesian, tree, shallow, deep, etc.) [169], and users must select the right strategy via domain knowledge or trial-and-error. For some ML models, none of the available strategies may be a good fit at all;

5. **Parallel performance tuning requires deep expertise**: a training algorithm’s optimal tuning parameters (e.g. batch size, step size) will change when going from single-machine to distributed execution, requiring trial-and-error to discover - for example, distributed training requires larger batch sizes to ensure high data sample throughput across multiple machines, which in turn necessitates different step sizes [26];

The chapter aims to address the aforementioned challenges 1-3, and defers a prototyping solution of Challenge 4-5 to the last part of this thesis.

Generalized from the thesis methodology presented from Chapter 3 to Chapter 4, in this chapter, we introduce a concept, *composibility* of ML parallelisms. In the context of data-parallel training, we demonstrate an end-to-end composable system, AutoDist, that allows mechanically composing model- and resource-adaptive distributed synchronization strategies from a few base, atomic synchronization aspects.

### 7.1 Introduction

Existing parallel DL systems, including those presented in previous chapters, have explored various aspects to improve the performance of synchronization, e.g. communication architectures [76, 87, 126], message computation and augmentation [54, 92], bridging models [22, 60], etc. These systems have demonstrated improved training speed or scalability on specifically-chosen DL “model architectures”, such as convolutional [58], fully-connected, but do not always generalize well to other architectures, or even models that incorporate multiple architectures [32]. However, from a practical usability standpoint, users are expected to choose and use the best system for their model of interest, which often requires both ML and systems savvy; or they may find that no system provides adequate parallel training performance for novel and complex models with multiple architectures, or substantial additional efforts are needed in tuning or augmenting existing systems to extrapolate to a new model.

To overcome the lack of flexibility of existing monolithic DL systems with built-in distributed strategies, we posit that such systems need to adopt the principle of *composability* – different aspects of performance-improving strategies, such as different choices of communication topologies (e.g., master-slave, peer-to-peer), update aggregation structures (ring, tree, hierarchy), and message encoding (e.g. “regular” stochastic gradients, sufficient factors [156]) can be mechanically selected and recombined to adapt to different building blocks of DL models [76, 169]. We refer to this approach as *strategy composition* for DL systems. The composition opens space to
improve the performance of synchronization.

**Exploiting the Structures of DL Models for Optimal Strategy Composition**

Because elements of a composite distributed-system strategy may apply deferentially to different building-blocks such as layers of a DL model, e.g. a parameter server \([19, 60, 87]\) for sparse gradients, and collective allreduce \([46, 126]\) for convolutional layers, we can exploit the structures of given algorithms or design novel algorithms that co-optimize multiple aspects and assign to every model building block, so to provide the greatest parallel throughput.

**Improving DL Training Systems with Task-adaptive Composability**

Existing distributed DL training systems are usually built around a single or only a few parallelization aspects, e.g. collective allreduce for Horovod \([126]\). Our experiments show that this approach is over-specialized, in the sense that no parallelization aspect we studied performs well across all DL models. The architecture of DL training systems is often tightly coupled with one aspect; thus, improving these systems’ performance on new models requires low-level modifications to their implementations. We see a need to break the design of DL training systems into composable units based on different aspects. We believe a modular design, based on these units, can help DL training systems adapt quickly to newly-emerging DL models.

**Developing Systems that Achieve High Performance on DL Models “Out-of-the-box”**

State-of-the-art DL models, such as BERT \([32]\), can incorporate multiple layer types of varying sizes connected in complex ways. In §7.6.2, we show that applying DL parallel training systems to complex models “out-of-the-box” (i.e. with default recommended settings) often results in lower-than-expected performance, when compared to reported results on typical benchmark models. While it could be argued that a skilled user can tune these systems and restore most of the “missing” performance, we think a better alternative is to develop automatic systems, which take advantage of strategy tuning algorithms to achieve high performance out-of-the-box. This would make parallel DL training more attractive and easier to use for a wider audience.

To explore these opportunities, we propose to design a composable and automated system for distributed parallel DL training. Most prior systems are built upon one (or a few) parallelization aspects, and apply those uniformly across all parts of a DL model (Figure 7.1(a)). In contrast, we aim to pursue a composable system that automatically generates a distributed strategy for a given model and resource specification (Figure 7.1(b)), by selecting from a pool of atomic parallelization aspects that can come from prior systems literature, and be extended as new research emerges. Such a system would exhibit the following obvious advantages:

**Unified Representation**

It possesses a *unified representation for synchronization* that jointly encodes multiple contributing factors to distribution performance (§7.3). With the representation, it go through a compilation process: it generates a suitable strategy conditioned on the user model and resource specification, taking into account the statistical properties of model building blocks, as the thesis
methodology *adaptive parallelism* suggests. The strategy is explicitly represented, executable by low-level systems, and transferable across users.

**Composable System**

The system allows strategy composition. Each aspect of parallelization maps to a primitive distribution function implementation (a.k.a. *kernel*) in the backend system. A complete distribution strategy is hence materialized by composing multiple primitive kernels, based on the strategy representation. This approach isolates strategy prototyping from low-level distributed systems, allows composition, assessment and execution of complex distribution strategies via a unified interface, and is extensible to emerging parallelization techniques.

**Automatic Parallelization**

The introduced representation spans a combinatorial space enclosing all possible strategies (i.e. assignments of specialized parallelization aspects to model building blocks). It allows to build a strategy auto-optimization pipeline (presented in Chapter 8) to efficiently optimize strategies against model and resources, which not only improves parallel performance, but is also an added convenience for users.

Following this discussion, we next ground our discussion and development in the context of data-parallel training, and derive the representations for *synchronization strategies*, and corresponded composable data-parallel training system – AutoDist.
Figure 7.1: How (a) prior DL training systems, and (b) a composable system differ in parallelizing an ML model.
7.2 Overview

Throughout this chapter, we will adopt the dataflow graph as the principled representation for ML models. We refer to the dataflow graph of DL models and its computational function using $D = \{(O, V_\Theta), \mathcal{E}\}$, where the node of $D$ is either a computational operation $o \in O$ or a stateful variable $v \in V_\Theta$, and edges ($\mathcal{E}$) are tensors. In this section, we focus on data-parallel training, which parallelizes $G$ over multiple devices following

$$V_\Theta^{(t+1)} \leftarrow V_\Theta^{(t)} + \epsilon \sum_{p=1}^{P} \nabla G_p(V_\Theta^{(t)}, X_p),$$

(7.1)

where the original single-device graph $G$ is replicated into $P$ replicas $\{G_p\}_{p=1}^{P}$, and each replica $G_p$ performs computation on the $p$th device over its independent split of training data $X_p$. All replicas use a consistent set of trainable parameters $V_\Theta$. In addition to the forward-backward computation, the core step making this possible is synchronization (mapping to $\sum$ in the Eq.), which collects parameter updates from multiple devices and maintains consistency on $V_\Theta$.

The synchronization step is expensive when training large models over many nodes [19, 60, 88]. Accordingly, many systems are developed to improve the performance of this synchronization step, each with its variety of synchronization aspects. Examples include communication architectures, topologies and bridging models [76, 126, 152, 169], partitioning and placement strategies [71, 100, 149], message encoding schemes [92], among others. However, multiple aspects interact with each other, together with complex structures in $G$ and $V_\Theta$, dictate their performance against different models and cluster specifications, thereby leading to an intricate space of synchronization strategy considerations. Existing strategies and systems usually treat $G$ and $V_\Theta$ as an entirety, and make specific instance choices for each aspect and ignore such intricacy. This presents opportunities that, though a unified strategy representation and a composable backend, the synchronization performance could be improved by co-optimizing multiples synchronization aspects against each building block of the model.

To formalize a generic representation for synchronization (§7.3), as a first step, we break existing instance-based synchronization methods into atomic units and re-catalog them into the following aspects:

- **Replication** of the model graph $G$: how and which device to replicate the model graph for data parallelism.

- **Partitioning and placement** of variables $V_\Theta$: partitioning, sharing and placement mechanisms of trainable variables.

- **Synchronization architecture and aggregation structures**: How to set up a network topology for message synchronization (e.g. bipartite parameter servers [60], P2P broadcast [156], or fully-connected all-reduce [126]). How and where to aggregate updates $\nabla G_p$, e.g. full summation at a central device [60], partial summation across a tree [75].

- **Message encoding**: How to encode and decode messages $\nabla G_p$ before communication, e.g. various forms of gradient updates, compression, sufficient factors [156], etc.
• Bridging models: How to bridge computation (of $\sum_{p=1}^{P} \nabla G_p$) and communication, e.g. with various synchronous, asynchronous, or bounded-asynchronous methods [24, 60, 152].

Figure 7.2: An example of how the synchronization strategy is composed based on the DL model $G$ and resource information in $R$.

Upon above, we denote $S$ as an expression (developed in §7.3) that instructs how $G$ should be synchronized when distributed on the cluster. We call it a synchronization strategy. It spans multiple synchronization aspects cataloged above, and is model-dependent and resource-dependent. Hence, a valid and complete $S$ would instantiate proper values for each aspect. We use $R$ to describe the specifications of the target cluster. It includes information such as the number of nodes, their addresses, number of CPUs and GPUs per node, and their interconnection information (e.g. Ethernet bandwidth). (Figure 7.1(b)).

AutoDist takes the dataflow graph $G$ and the cluster specification $R$ as inputs. It makes an appropriate choice for each aspect in its corresponded representation space (§7.3), w.r.t $G$ and $R$, considering the computational or statistical properties present in $G$ and the resource condition in $R$. It then composes them into a complete strategy expression $S$. Figure 7.2 illustrates such an example.

AutoDist decouples the strategy generation from specific system implementation. The generation of $S$, depending on user choices, is done either via hand-designed strategy builders, or an AutoStrategyBuilder that derives the optimal $S$ via automatic strategy optimization (Chapter 8). For instance, we can have a strategy builder for Horovod [126] that takes $G$ and $R$ to build a strategy with collective allreduce as the communication architecture, ring as the reduction structure.

The expression is then broadcasted across all corresponded workers within $S$. An AutoDist process then on each worker is invoked locally to transform the graph $G$ based on $S$, into a distributed graph. AutoDist backend implements a library of primitive and generic graph trans-
formation functions, which we will call graph transformation kernels. By mapping each aspect in $S$ with its graph transformation kernel, the transformation happens in a composable way, by applying a series of kernels instructed in $S$. Upon the completion of graph transformation, AutoDist launches the specific ML frameworks with the desired distributed dataflow graph to start distributed training.

We next develop each component of $S$ in detail.

### 7.3 Synchronization Strategy Representation

The synchronization expression $S$ defines the space for high-level strategy generation and serves as a composition instruction for low-level strategy-based graph rewriting (§7.4). We define the synchronization representation with two levels of semantics: graph level that expresses how $G$ will be transformed globally to fit a parallel environment, and node level that specifies per-variable synchronization configurations. Figure 7.3(a) illustrates an example of strategy expression.

#### 7.3.1 Graph-level Representation

At the graph level, we introduce the graph config to include a single semantic, replication, to capture the aspect of how $G$ should be replicated across multiple devices in data-parallel environment. We use a tuple $(G, \{d_i\}_{i=1}^m)$ to notate that $G$ will be replicated on a set of destination devices $\{d_i\}_{i=1}^m \subset R$.

#### 7.3.2 Node-level Representation

At the node level, for each trainable variable $v_i \in V_\Theta$, we introduce a node config to express its variable-specific synchronization setup, specifying the following aspects.

**Variable Partitioning**

Operation partitioning in dataflow graphs is a widely explored topic in parallel DL systems [71, 128, 149]. As a complement to existing works that propose a general partitioning mechanism for arbitrary operations, AutoDist only concerns the partitioning of nodes that participate in data-parallel synchronization, i.e. nodes in $V_\Theta$. Since a variable node could be partitioned directly along any axis of its tensor buffers (unlike an arbitrary computational operation), we represent the partitioning of a variable $v_i$ simply as a vector $p_i = [p^j_i]_{j=1}^{k_i}$, with $k_i$ equals the number of dimensions of $v_i$, and $p^j_i$ an integer representing how many partitions are generated along the $j$th axis of $v_i$. We thus obtain $\prod_{j=1}^{k_i} p^j_i$ partitioned variables of $v_i$ after the partitioning. The representation allows to express a rich set of optimizations in existing synchronization systems, such as various types of sharding strategies for load balancing in PS [87, 115], but is much more flexible as it is variable-specific.
Figure 7.3: A composed strategy (§7.3) and its mapped graph-rewriting operations. A solid arrow is from a tensor producer to a stateless consumer; dotted refers to a stateful variable update. (b) It shards variable $W$ into $W_0$, $W_1$ along axis 0. (c) It replicates onto two devices. (d) $W_0$ is synchronized by sharing the variable at one device and applying $(\text{Reduce}, \text{Broadcast})$, $W_1$ is synchronized with $\text{AllReduce}$. 
Variable Sharing and Placement

A variable, partitioned or unpartitioned, can be shared (e.g. PS), or replicated across multiple devices (e.g. collective communication) for synchronization. We bring in placement that decides where the variable locates at. When it is instantiated as a set of devices $\{d_i\}_{i=1}^m \subset \mathcal{R}$, the variable is replicated across all $\{d_i\}_{i=1}^m$. A single device value $v_i : d_i$ then indicates $v_i$ is placed on $d_i$ and the computation or synchronization happening on different devices have shared access to $v_i$. While device placement can be extended onto all nodes in $\mathcal{G}$ [100], AutoDist only models the placements of $V_\Theta$, as the rest of nodes in $\mathcal{G}$ are replicated across $\mathcal{R}$ in data-parallel training, captured by the introduced replication semantic in graph config.

Synchronization

Data-parallel synchronization requires collecting and applying variable updates $\nabla \mathcal{G}_p$ from distributed replicas, and synchronizing all replicas with new states of $v_i$. This distinguishes synchronization form single-directional communication between devices, as normally addressed in model-parallel systems [71, 149]. Specific to this characteristic, for each variable $v_i$, we introduce a synchronizer config to capture the process, which has its leading dimension as synchronizer type, indicating the communication primitives used for synchronization, notated as combinations of communication primitives. For example, $v_i$ with synchronizer type as ($\text{Reduce}$, $\text{Broadcast}$) and placement as $d_i$ follows a bipartite PS architecture to reduce and apply the updates $\{\nabla \mathcal{G}_p\}_{p=1}^P$ onto the destination $d_i$, and broadcast the updated states back to all participating replicas. Similarly, type ($\text{AllReduce}$) straightforwardly indicates using AllReduce for joint aggregation and synchronization. This forms a unified representation of diverse synchronization architectures.

Based on the synchronizer type, we bring in more type-specific semantics such as: (1) transfer destination: $d_t (d_t \in \mathcal{S})$, to express a two-level hierarchy to reduce updates or broadcast parameters (of $v_i$) first to the transfer station $d_t$, then to the final destination $d_i$. (2) reduction sequence: list of devices $(d_1, \ldots, d_n)$ indicating the order of devices when performing collective communication. (3) group: schemes to group multiple communication primitives with the same group, together into one single operation to reduce constant min-cost (see §7.6.2 for a thorough study). The offers flexibility to specify, for each $v_i$, various forms of communication topologies, aggregation structure, grouping mechanisms (e.g. ps, hierarchical ps, ring or tree reduce), and related optimization conditioned on the context of $v_i$ in $\mathcal{G}$ and the resource $\mathcal{R}$.

Message Encoding and Decoding

Many synchronization systems exploit certain structures (e.g. low-rank) present in the original message $\nabla v_p$, and encode them into alternative forms for faster network transfer. We introduce encoder and decoder for each $v_i$ to model this aspect. Optimizations like sufficient factor broadcasting [156], or compression [92] hence can be covered by specifying each variable with a corresponding encoder and decoder methods in its node config.

\footnote{A sparse version of it has the synchronizer type ($\text{Gather}$, $\text{Scatter}$).}
Bridging Models

This aspect specifies how far the parallel replicas can allow computations and messages to occur out-of-order. Based on previous studies [26, 60] and definitions, we use an integer \textit{staleness} to express the degree of consistency for synchronizing the variable \( v_i \) – when staleness is set to \( K \), the fastest worker replica cannot be more than \( K \) steps ahead of the slowest worker in Equation 7.1.

7.3.3 Expressiveness

Table 7.1 enumerates several notable synchronization systems and shows how their specialized optimizations can be represented by the proposed synchronization representation. Since AutoDist focuses on synchronization in data-parallel training, the proposed representation deliberately skips the support for the partitioning of computational operations. Due to our assumptions on the synchronization aspect be represented using dataflow graph rewriting, the proposed representation also excludes optimizations commonly done in lower execution layer of the system, such as scheduling [55, 115, 169] and memory management [15, 23]. Based on this representation, we next present AutoDist system design to enable such strategy composition.
<table>
<thead>
<tr>
<th>System</th>
<th>Synchronization Architecture</th>
<th>Variable Sharding</th>
<th>Variable Placement</th>
<th>Collective Fusion</th>
<th>Encoding</th>
<th>Reduction Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poseidon [169]</td>
<td>(Reduce, Broadcast) for normal gradients, (Broadcast) for rank-1 gradients</td>
<td>Fixed-size partitioning for all $V_{\mathcal{G},\theta}$</td>
<td>Place on PS</td>
<td>N/A</td>
<td>Sufficient factor</td>
<td>Local CPU as transfer device</td>
</tr>
<tr>
<td>Parallax [76]</td>
<td>(Gather, Scatter) for sparse gradients, (AllReduce) for dense gradients</td>
<td>Fixed-size partitioning for all sparse $v \in V_{\mathcal{G},\theta}$</td>
<td>Placed on PS for sparse $v$, replicated and placed on all worker nodes for dense $v$</td>
<td>N/A</td>
<td>N/A</td>
<td>Local CPU as transfer device for shared var, and ring for AllReduce.</td>
</tr>
<tr>
<td>Horovod [126]</td>
<td>(AllReduce) for dense gradients, (AllGather) for sparse gradients</td>
<td>N/A</td>
<td>Replicated and placed on all worker nodes</td>
<td>fixed-size fusion</td>
<td>FP16 compressor</td>
<td>Specifying device order (tree, ring, etc.)</td>
</tr>
<tr>
<td>BytePS [115]</td>
<td>(Reduce, Broadcast) for all gradients</td>
<td>Fixed-size partitioning for all $V_{\mathcal{G},\theta}$</td>
<td>Placed on PS</td>
<td>N/A</td>
<td>FP16 compressor</td>
<td>Local CPU as transfer device</td>
</tr>
</tbody>
</table>

Table 7.1: The proposed strategy space in AutoSync covers many advanced strategies in existing systems. Moreover, it offers a superset of semantics beyond them.
7.4 AutoDist: Composable System via Graph Rewriting

AutoDist backend is based on *dataflow graph rewriting*. Given user-defined $G$, AutoDist incorporates distribution semantics by adding, deleting or modifying nodes and edges and their attributes in $G$, using designated graph transformation kernels. Afterward, the transformed graph with distributed semantics is submitted to the runtime for execution.

7.4.1 Strategy Verification

Before a strategy can be applied to $G$ for rewriting, it goes through a light compilation process for verification. The process looks for incompatibilities, such as reduction destinations out of $R$ or inconsistent with the variable placement, or combinations of communication primitives that do not guarantee synchronization of variables, etc.

7.4.2 Graph Rewriting Kernels

AutoDist backend offers a library of primitive graph rewriting kernels. Each sub-expression in $S$ is mapped to a kernel type with specific configurations. Each type of kernel defines how to rewrite the graph from its current state to the next state, which manifests the corresponding semantics in the sub-expression. By designing each local kernel at the right level of abstraction, they can be flexibly composed to alter the graph based on $S$. We describe a few representative kernels illustrated with illustrations in Figure 7.3.

**Partitioner** (Figure 7.3(b)) reads node-level partitioning configurations from $S$ for each $v_i \in V_\Theta$. It splits the variable across given axes into multiple smaller variables, as well as its gradients and subgraphs corresponded to its state-updating operations. However, it does not split the operations that consume the original variable – which will instead consume the value re-concatenated from all partitions. Without allowing recursive partition, each of the new smaller variables has its own node config (generated at strategy generation time), will be added into $V_\Theta$ as an independent variable identity in the following transformation steps. **Replicator** (Figure 7.3(c)) reads graph-level configuration from $S$. It replicates the original graph onto target devices. Unless overridden by other graph-transformation kernels or by developers, the replicated operations or variables have their placement same with the target replication destination in $S$. **Synchronizer** (Figure 7.3(d)) reads node-level configurations for each of the original and partitioned variable in $V_\Theta$, where **Compressor** as its component rather than graph-transformation kernel is responsible for gradient encoding and decoding therein. Depending on the synchronizer type, it rewrites the graph: (i) either to share a variable on a destination device across replicas ($(\text{Reduce, Broadcast})$ synchronizer) with specified staleness in $S$, (ii) or to synchronize states of replicated variables via collective communication (AllReduce synchronizer) following specified device structures in $S$. Moreover, the kernel implementations are dispatched to handle either dense or sparse cases.

Besides existing kernels, the system design allows convenient extensions to emerging synchronization optimizations, by allocating new dimensions in the representation and introducing corresponded graph-rewriting kernels in the backend.
7.4.3 Parallel Graph Rewriting

With a valid $S$ and graph-transformation kernels, parallelizing the ML program is essentially applying a series of kernels to rewrite the graph $G$ into a distributed graph $G'$. In contrast to other graph optimization systems [16, 72, 84], direct generation of $G'$ leads of the size of $G'$ proportionally growing with both the size of $G$ and $R$, quickly becoming unmanageable in terms of memory or rewriting efficiency if any of $G$ and $R$ is large. AutoDist uses local and deferred graph rewriting to mitigate the issue. Instead of writing all distributed semantics in one graph, AutoDist first broadcasts the strategy expression (whose size only corresponds to $G$) across $R$, and invokes graph transformation on each node afterward. Each node then transforms the single-node graph in parallel, but only rewrites a local subgraph of $G'$ that corresponds to graph execution on that node. They maintain a consistent picture of $G'$ by checking against $R$ and $S$, and uses absolute device notations when generating shared or distributed nodes. This allows to scale to extremely large graphs or clusters. After transformation, the executions of the dataflow graph is delegated to specific framework runtime on each node, so AutoDist keeps agnostic to frameworks. More implementation details are in §7.4.4.

7.4.4 Implementations

AutoDist is built as a Python library (12K LoC), and relies on specific ML frameworks to provide distributed runtime for dataflow graph execution. The current AutoDist offers APIs compatible with TensorFlow [3], but the design and concepts are compatible with other frameworks.

Coordination

In addition to described components, AutoDist implements a cluster coordinator. It sets up connections from the launching node, denoted as chief, to each worker specified in $R$. After chief generates a strategy, it broadcasts the serialized strategy across nodes, then coordinates each worker to apply the graph rewriting based on the strategy to $G$, and starts the framework process to execute the transformed graph.

Usages

AutoDist interface is designed to make sure ML practitioners can write distributed ML code as if they were writing single-node code. Using AutoDist involves construct a desired strategy builder with a resource specification, and then properly scopes (or decorates) the model declaration code using the provided interface, so that AutoDist can capture the corresponded dataflow graph representation for upcoming strategy generation. Then one can use the AutoDist session instead of the framework session to execute the graph. Figure 7.4 illustrates a code example.

Developing a synchronization strategy in AutoDist is equivalent to implementing a custom strategy builder. In AutoDist, this can be done using pure high-level languages (e.g. Python), without modifying low-level distributed system implementations. For emerging and new synchronization technologies, adding their support in AutoDist requires implementing their corresponded graph transformation kernels, and registering them as new aspects in strategy representation so as to make them available in strategy generation.
import autodist
ad = autodist.AutoDist(resource=spec, strategy=builder='AUTO')
with o.scope():
    # Build your model here
    fetches = ...
    feeddict = ...
    o.Session().run(fetches, feeddict)

Figure 7.4: An code snippet illustrating how to use AutoDist to distribute single-node ML code.

### 7.5 Automatic Strategy Optimization

A unified and explicit representation of the synchronization strategy and a ready-to-evaluate system enables to auto-optimize $S$ conditioned on $G$ and $R$, formulated as

$$
\max_S u(S, f(G, R, S), R),
$$

(7.2)

where $f(\ast)$ denotes the transformation by AutoDist, and $u$ characterizes the system throughput that we want to maximize\(^4\) Solving Equation 7.2 automatically alleviates the knowledge barriers on selecting appropriate distributed strategies. AutoDist builds a strategy auto-optimization pipeline, AutoSync, to efficiently optimize strategies against model and resources, which not only improves parallel performance, but is also an added convenience for users. The optimizer is built on top of both principled system modeling and a data-driven ML models trained on low-shot trial-run data, and can improve as more data is acquired.

We defer the detailed development of the solutions to Equation 7.2, and the evaluation of the solution, to Chapter 8. We focus on evaluation of the system composability in AutoDist in the next section.

### 7.6 Evaluation

In this section, we evaluate the AutoDist system, aiming to answer the following questions:

- How do multiple aspects of synchronization influence the data-parallel training performance? Is a composable design of synchronization system necessary?
- Is a composable design of data-parallel synchronization system feasible? How does the system perform when representing existing strategies?

#### 7.6.1 Experiment Setup

**Clusters**

We focus on GPU clusters since it is the main setup for distributed DL training. We conduct experiments on two setups:

\(^4\)It is possible to optimize other unities, e.g. convergence.
Cluster Setup

<table>
<thead>
<tr>
<th>R</th>
<th>Cluster Setup</th>
<th>GPU Distribution</th>
<th>Bandwidth Spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>16x Cluster A nodes</td>
<td>[1] x 16</td>
<td>40GbE</td>
</tr>
<tr>
<td>A2</td>
<td>11x Cluster A nodes</td>
<td>[1] x 11</td>
<td>40GbE</td>
</tr>
<tr>
<td>B1</td>
<td>2x g3.16</td>
<td>[4] x 2</td>
<td>25 GbE</td>
</tr>
<tr>
<td>B2</td>
<td>3x g3.16</td>
<td>[4] x 3</td>
<td>25 GbE</td>
</tr>
<tr>
<td>B3</td>
<td>4x g3.16</td>
<td>[4] x 4</td>
<td>25 GbE</td>
</tr>
<tr>
<td>B4</td>
<td>1x g4dn.12</td>
<td>[4] x 1</td>
<td>50 GbE</td>
</tr>
<tr>
<td>B5</td>
<td>2x g4dn.12</td>
<td>[4] x 2</td>
<td>50 GbE</td>
</tr>
<tr>
<td>B6</td>
<td>3x g4dn.12</td>
<td>[4] x 3</td>
<td>50 GbE</td>
</tr>
<tr>
<td>B7</td>
<td>4x g4dn.12</td>
<td>[4] x 4</td>
<td>50 GbE</td>
</tr>
<tr>
<td>B8</td>
<td>8x g4dn.12</td>
<td>[4] x 8</td>
<td>50 GbE</td>
</tr>
<tr>
<td>B9</td>
<td>1x g3.4, 1x g3.16</td>
<td>[1, 4]</td>
<td>10/25 GbE</td>
</tr>
<tr>
<td>B10</td>
<td>1x g3.16, 1x g4dn.12</td>
<td>[4] x 2</td>
<td>25/50 GbE</td>
</tr>
<tr>
<td>B11</td>
<td>2x g3.16, 2x g4dn.12</td>
<td>[4] x 4</td>
<td>25/50 GbE</td>
</tr>
<tr>
<td>B12</td>
<td>1x g4dn.2, 1x g4dn.12</td>
<td>[1, 4]</td>
<td>25/50 GbE</td>
</tr>
</tbody>
</table>

Table 7.2: Cluster specifications we have experimented with, listed with their reference name (R), setup information, GPU distributions, and bandwidth specifications.

- **Cluster A** includes maximally 16 nodes, each equipped with a GeForce TITAN X GPU, an Intel 16-core CPU and 64GB RAM, interconnected via a 40-Gigabit Ethernet switch;

- **Cluster B**, based on AWS, consists of up to 8 nodes, each node is one of the g3.4xlarge (1x Tesla M60 GPU and 10GbE Ethernet), g3.16xlarge (4x M60, 25GbE), g4dn.2xlarge (1x NVIDIA T4 GPU and 25GbE Ethernet), g4dn.12xlarge (4x T4, 50GbE) instance types. Due to AWS constraints, they all have 10GbE single-flow bandwidth. On top of these two clusters, we list all the resource specifications we have experimented in Table 7.2 for later reference.

We rely on TF 2.0 for both dataflow graph evaluation and distributed execution, and *do not alter* any runtime or communication libraries in the native TF. As a note, TF 2.0 is complied with CUDA10.0, CUDNN 7.1, and NCCL 2.4.7, and uses gRPC for network communication. We choose baselines that depend on the same runtime in later sections.

**Benchmark Models and Metrics.**

Following MLPerf [97], we choose a diverse set of ML models listed in Table 7.3 for evaluation. We focus on the *system throughput*, and report *per-iteration training time*, instead of statistical convergence to evaluate the system. We conduct fully synchronous training, and exclude synchronization aspects that would change the nature or results of the ML training as in the original single-node program. We manage to train all benchmark models to the reported accuracy [97] – hence we skip this comparison in later sections.
<table>
<thead>
<tr>
<th>Model</th>
<th>Task</th>
<th>Training data</th>
<th>Batchsize</th>
<th>#Params</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet101 [57]</td>
<td>IC</td>
<td>ImageNet</td>
<td>32</td>
<td>45M</td>
</tr>
<tr>
<td>InceptionV3 [139]</td>
<td>IC</td>
<td>ImageNet</td>
<td>32</td>
<td>24M</td>
</tr>
<tr>
<td>VGG16 [129]</td>
<td>IC</td>
<td>ImageNet</td>
<td>32</td>
<td>138M</td>
</tr>
<tr>
<td>Densenet121 [63]</td>
<td>IC</td>
<td>ImageNet</td>
<td>32</td>
<td>8M</td>
</tr>
<tr>
<td>Transformer [145]</td>
<td>MT</td>
<td>WMT’14 ende</td>
<td>5K</td>
<td>62M</td>
</tr>
<tr>
<td>NCF-dense [59]</td>
<td>CF</td>
<td>MovieLens-20Mx16x32 [97]</td>
<td>256</td>
<td>122M</td>
</tr>
<tr>
<td>BERT-3L [32]</td>
<td>LM</td>
<td>Wiki &amp; BookCorpus</td>
<td>32</td>
<td>11M</td>
</tr>
<tr>
<td>BERT-6L [32]</td>
<td>LM</td>
<td>Wiki &amp; BookCorpus</td>
<td>32</td>
<td>36M</td>
</tr>
<tr>
<td>BERT-12L [32]</td>
<td>LM</td>
<td>Wiki &amp; BookCorpus</td>
<td>32</td>
<td>110M</td>
</tr>
<tr>
<td>BERT-large [32]</td>
<td>LM</td>
<td>Wiki &amp; BookCorpus</td>
<td>8</td>
<td>340M</td>
</tr>
</tbody>
</table>

Table 7.3: ML models we have experimented with. All the model implementations are from the tensorflow/models repository. IC: image classification, MT: machine translation, LM: language modeling. For neural collaborative filter (NCF), we follow MLPerf [97] and use an enlarged version (x16x32), and we use dense instead of sparse gradients for its embedding variables to test systems’ capability.

### 7.6.2 Study of Synchronization Aspects

In this section, we re-examine several synchronization aspects commonly adopted in the latest development.

#### Synchronization Architecture

When choosing synchronization architectures, seemingly contradictory conclusions have been drawn in recent literature, claiming the supremacy of one architecture over another, e.g. Parallax [76], based on NCCL, shows a constant improvement of using collective communication to synchronize dense gradients over PS. ByteScheduler [115], using a different distributed runtime, demonstrates PS with credit-based scheduling is better.

Based on the native TF distributed runtime and NCCL, we compare the two architectures on a series of dense models across different resources shown in Figure 7.5, using the three builders TF-PS-LB, AR and PS (§7.6.3). On BERT-3L, AllReduce demonstrates outstanding advantage (more than 2x faster) over both unpartitioned (PS1) and partitioned PS (PS2), but when we increase the model size to BERT-large, partitioned PS outperforms AR by a small margin. On another model NCF-dense, the ranks between PS2 and AR change depending on the cluster. Both BERT and NCF have embedding variables that are significantly larger than other variables, preventing a normal PS without variable partitioning to achieve load balancing. This disadvantage no longer holds on ResNet101-B3 and Tranformer-B12, where PS1 shows small advantage over PS2. ResNet101 has most of its parameters as small convolutional filters, hence balancing them across multiple nodes is possible without partitioning, skipping overheads introduced by splitting and concatenation. Comparing Transformer-A1 and Transformer-B2, we observe AllReduce betters PS significantly on 16 nodes (A1), but underperforms PS substantially if switching to B2 (0.3s increase on per-iter time). We also perform experiments on two models with sparse gradi-
Figure 7.5: Comparing synchronization architectures across a diverse set of models and cluster specifications. PS1: load balanced PS builder without partitioning, PS2: load balanced PS builder with partitioning. AR: collective AllReduce based builder. Per-iter training time is reported.

ents: NCF-sparse and LM1B [10] and the results echo the advantages of PS over AllGather on synchronizing gradient with sparsity above a certain level – this is modeled and supported by the proposed synchronization presentation.

In summary, we see that even for synchronizing dense gradients, different architectures seem to demonstrate uncertain pros and cons, depending on models and resources.

Collective Communication Merge Scheme

A common optimization in collective-based synchronization is to merge multiple collective operations into a fused one, to reduce the linear min-cost by the invocation of each collective. While this optimization has been adopted broadly [84, 126], its effects and how to properly tune it against different models remain largely unknown. To study it, we implement a AllReduce(chunk: int) builder with a chunk-based merging scheme, similar to tf.distribute. Under this scheme, starting from the first variable, the synchronization of every $n = \text{chunk}$ adjacent variable will be merged into one collective. Based on this builder, we experiment with different values of a chunk, on diverse model-resource combinations shown in Figure 7.6.

We observe the chunk effects differently on different models – among the values we examined, the performance improvement due to collective merging can go up to more than 2x (ResNet101), or with almost no impact (VGG16). On NCF-dense, we observe merging undermines the performance. We hypothesize this is because NCF-dense uniquely has two adjacent large embedding variables; communicating them in one collective might be bandwidth-bounded – in this situation, partitioning them might be a better choice. Constantly, the performance drops
When the chunk size is larger than the number of variables in the model. Merging all variables into one collective operation prevents opportunities of pipelining communication with computation during BP, which is a key optimization nowadays distributed runtime has adopted. In choosing the optimal value of chunk, we cannot draw a constant conclusion: while on BERT-6L, BERT-12L, Transformer, 3 transformer-based architectures, this value seems to be near 64, it shifts toward 128 on BERT-large. On DenseNet121, we see a sudden drop of per-iter time when the chunk goes up to 512, but the performance of VGG16 is not influenced by it. Chunk-based merging is obviously suboptimal since it is inflexible and not fully aware of the model structure. How to create the optimal merge scheme that copes with the model structure is an NP problem, explored in Chapter 8.

To conclude, the optimal choice of each aspect changes with models, resources, and the choice of other aspects, letting alone the fact that there exists variability caused by system artifacts. Understanding and interpreting such intrinsic system-ML subtleties are nontrivial. We do
not think a typical ML practitioner should be reasonably expected to have this knowledge, and we argue this demonstrates the need for model- and resource-dependent modeling of synchronization strategies, and corresponding auto-optimization methods.

### 7.6.3 Hand-composed Strategy Performance

In this section, we implement a few notable synchronization strategies as fixed strategy builders in AutoDist, and compare them to those specialized systems in terms of system throughput. We conduct all the experiments on Cluster B (B4-B8). We consider the following baselines:

1. **MirroredStrategy** where each “strategy” from the `tf.distribute` library can be seen as a fixed strategy builder based on specialized dataflow graph writing. The mirrored strategy is based on ring allreduce⁵;

2. **TF-PS-LB** where the native TF-PS is reported to have major drawbacks in load balancing [115, 169]. We improve it with a greedy load balancing strategy – we maintain a load recorder, and place each variable on the next PS which has the least loads. With this enhancement, we have observed much stronger performance under TF2.0 distributed runtime – which we will use as a PS baseline;

3. **Horovod**: We deploy Horovod 0.19.0 with NCCL 2.4.7. It is a strong collective communication baseline specialized in using `AllReduce`, `AllGather` for parameter synchronization;

4. **AllReduce builder**: We implement an `AllReduce` builder in AutoDist which exactly reproduces the parameter synchronization in Horovod. Related hyperparameters for each model (e.g. merge scheme, see §7.6.2) is hand-tuned and the best performance is reported.

5. **PS builder**: Manually implemented parameter server builder with many known optimizations incorporated: it performs hierarchical `Reduce` and `Broadcast` [87] on nodes with ≤1 GPUs to reduce network traffic; it partitions large variables (upon a hand-tuned size threshold based on model) into number of partitions equal to number of nodes in $S$, and evenly distributes shards on nodes to achieve load balance [76, 115]. We use it as a strong PS baseline based on TF distributed runtime.

We compare these methods on 4 benchmark models: BERT-large, DenseNet121, ResNet101, InceptionV3, in Figure 7.7. We observe that collective-based strategies outperform the rest by a large margin, on all single node settings (1, 2, 4 GPUs), in which settings NCCL is advantageous. The improved version of TF parameter server, TF-PS-LB shows constant and acceptable performance other than reported [76, 115]. It scales well on CNNs (DenseNet121, InceptionV3, and ResNet101), outperforms Horovod occasionally in several settings (unlike reported), which verifies TF distributed runtime is a valid testbed. All baselines seem to scale poorly or sublinearly on BERT-large, among which PSBuilder performs best.

⁵Among all strategies we only managed to setup the MirroredStrategy running on up to 4 GPUs on a single node during the writing of this paper.
The reconstructed AllReduceBuilder shows almost similar performance to Horovod on all CNNs, and slightly worse performance on BERT-large though, probably because Horovod has a smarter mechanism\(^6\) for collective communication fusion. The hand-optimized PSBuilder frequently outperforms all other methods in distributed settings (≥4 GPUs), particularly evident on BERT-large. We will show different ranks between collective communication and PS in §7.6.2 later.

To summarize, using distributed TF as runtime, we show that the composed hand-optimized strategies in AutoDist exhibits at least matched performance with baseline systems, and offers the best all-round performance on all models, thanks to its flexibility to represent strategies as builders.

\(^6\)https://github.com/horovod/horovod/blob/master/docs/autotune.rst
Figure 7.7: Based on TF distributed runtime, we comparing MirrorStrategy, TF-PS-LB, Horovod with manually implemented AllReduce and PS builders on AutoDist. See §7.6.3 for a detailed analysis.
7.7 Additional Related Work

Specialized Synchronization Systems

GeePS [23] proposes a specialized PS for GPU clusters; Poseidon [169] and Parallax [76] use hybrid communication architectures exploiting specific properties present in the model. Horovod [126] brings in collective communication to reduce gradients for distributed DL, and a line of collective communication-based systems [46, 67, 161] have reduced the training time of specific NNs (e.g. ResNet) from days to minutes on specialized hardware. Most of these systems are specialized based on one or two synchronization aspects or hardwares, and exhibit varying performance on different models or resources.

Synchronization performance can also be optimized at lower levels of systems – such as better scheduling to pipeline computation and communication [55, 56, 115, 169]. While AutoDist does not include these optimizations as it assumes each aspect to map with a dataflow graph rewriting function, its representation and auto-optimization techniques can be integrated with and benefit from these system runtimes.

DL Graph Rewriting and Optimization

An emerging line of work, such as TVM [16, 17], TASO [72], XLA [44], perform automatic operator optimization and graph rewriting on a single device, mostly for inference graphs. AutoDist draws insights from these work but faces different challenges: the training graph needs to handle states, and the synchronization in data parallel training has unique semantics and problems.

Representation of Parallelisms

Some existing works [71, 100, 128, 149] have studied the representation of parallelisms in different context, and propose automatic parallelization methods. Device placement [100] automatically places nodes of a graph using a trained NN. Mesh-TensorFlow [128] defines “meshes” to describe how operations are partitioned and dispatched across devices. FlexFlow [70, 71] proposes the “SOAP” representation, and uses a stochastic MCMC-based algorithm to search for the optimal partitioning strategies for coarse-grained NN layers. Tofu [149] poses a similar problem on finer-grained operators, and derives possible partitioning configurations using symbolic interval analysis, and automatically optimize the partitioning. These techniques focus on partitioning or placement, and partially intersect with AutoDist as variables in the dataflow graphs are also represented as nodes (hence can be partitioned or placed using their techniques). However, AutoDist focuses on how to characterize the problem of synchronization in data-parallel training and modeling various synchronization-related aspects, and aims to co-optimize them to maximize the training performance.
Part III

Automatic ML Parallelization
Introduction

Echoing the fourth and fifth challenge mentioned in the opening of Chapter 7 — ML scale-up is frequently underestimated. What does it really take to train an ML model, whose training code was originally written for a single CPU/GPU, on multiple machines – The need to choose the appropriate distributed strategies, and heavily tune the distributed code to the satisfying system or statistical performance, which is yet another iterative process in addition to model development.

Built on top of the composable representation and system, this part of the thesis presents automation mechanisms for ML parallelization, to lower the barrier of distributed ML.

In Chapter 8, based on the synchronization strategy representation developed in Chapter 7 and the system AutoDist, we build an end-to-end pipeline, AutoSync, to automatically optimize synchronization strategies with ML-based simulators given model structures and resource specifications, lowering the bar for data-parallel distributed ML. By learning from low-shot data collected in only 200 trial runs, AutoSync can discover synchronization strategies up to 1.6x better than manually optimized ones. We develop transfer-learning mechanisms to further reduce the auto-optimization cost – the simulators can transfer among similar model architectures, among similar cluster configurations, or both. We also present a dataset that contains over 10000 synchronization strategies and run-time pairs on a diverse set of models and cluster specifications.

The results presented in this part of the thesis have also appeared in the following papers:

Chapter 8

Automatic Synchronization Strategy Optimization

8.1 Introduction

As shown in Chapter 7, existing systems struggle to provide excellent all-round performance on diverse models due to their oversimplified assumptions about the synchronization, and rigid application of fix-formed synchronization strategies (e.g. parameter server (PS) [87, 152] for BytePS [115], Allreduce for Horovod [126]), ignoring the characteristics of models or clusters. More importantly, different strategies often exhibit sharp performance differences when applied to different ML building blocks (shallow, deep, sparse, dense, etc.) [76, 169], and the burden of selecting the right strategy for the model of interest is placed on ML practitioners, who may not have domain expertise on the trade-offs among these systems. Given the combinatorial number of choices for various synchronization factors, (e.g. architecture, variable partitioning, and placement configuration), it is prohibitively costly to manually search for the optimal strategy, and the search has to be redone every time a new model is developed.

To address these challenges, this chapter aims to answer: Can one automate the selection of the optimal synchronization strategy, given a model and cluster specification? With multiple synchronization-affecting factors identified in Chapter 7 for data-parallel distributed DL, We construct a valid and large strategy space spanned by multiple factors, by factorizing the strategy with respect to each trainable building block of a DL model. To efficiently navigate the space and locate the optimal strategy, we build an end-to-end pipeline, AutoSync. AutoSync leverages domain knowledge about synchronization systems to reduce the search space, and is equipped with a domain adaptive simulator, which combines principled communication modeling and data-driven ML models, to estimate the runtime of strategy proposals without launching real distributed execution. To further reduce practical development cost, we study the transferability of trained simulators across different models and resource specifications, which shows promising adaptability to unseen models or cluster configurations.

We evaluate AutoSync on a broad set of models and clusters, and show that there exist ample strategies in the proposed space that outperform hand-optimized systems by a significant margin. AutoSync can effectively find strategies that reduce the training time by 1.2x - 1.6x than hand-
optimized ones on multiple model architectures (e.g. NCF [59], BERT [32] and VGG16 [129]),
within an acceptable budget. Leveraging transfer learning, AutoSync simulators can be trained
on cheaper trial data collected on smaller models or clusters, and used to derive strategies without
additional training for larger models or costlier clusters. As an additional contribution, we collect
a dataset with over 10000 data points containing (model, resource, strategy) tuples and their
corresponding runtime on real clusters. We share the dataset with the community to encourage
extended studies.

8.2 Problem Formulation

8.2.1 Notations

We represent a DL model using its dataflow graph \( G = \{(V_{G,\theta}, V_{G,o}), E_G\} \) where \( V_G \) are nodes
in \( G \) including trainable variables \( V_{G,\theta} = \{v_i\}_{i=1}^{V_G,\theta} \) or computational operations \( V_{G,o} \), and \( E_G \) are
tensors (edges) transferred between nodes. For simplicity, we use \( V \) equivalently with \( V_{G,\theta} \) to
notate the set of variables. In addition to \( G \), we define a cluster as a device graph \( D = \{V_D, E_D\} \),
where \( V_D = \{d_p\}_{p=1}^{V_D} \) represents devices (e.g. CPUs or GPUs), and \( E_D = \{b_{i,j}\} \) is a symmetric
matrix with the entry \( b_{i,j} \) representing the connectivity (e.g. bandwidth) between \( d_i \) and \( d_j \). In
data-parallel training, we replicate \( G \) on all devices, and update each trainable variable \( v_i \) using
the aggregation of the stochastic gradients \( \nabla v_i(G, X_p) \) computed by each worker device \( d_p \) on its
data partition \( X_p \), following
\[
v_i^{(t+1)} = v_i^{(t)} + \epsilon \sum_{p=1}^{P} \nabla v_i^{(t)}(G, X_p), \quad \text{for } v_i \in V_{G,\theta}.
\]
Since devices are distributed across the cluster, obtaining the aggregation requires synchronization support, which collects updates \( \nabla v_i^{(t)} \) and provides all devices the shared access to a consistent version of \( v_i^{(t+1)} \).

Existing systems aim to optimize some individual factor to expedite synchronization, ignoring
that the optimal of each factor significantly changes with \( G \) and \( D \). For ML practitioners, it is
challenging to select appropriate synchronization strategies for their models of interest without
domain expertise.

8.2.2 Formulation

Alternatively, we pose the strategy selection as an optimization problem, in which the per-
iteration runtime (e.g. time taken to process a batch on all nodes of the entire cluster, equivalent
with system throughput), denoted as \( f \), is minimized given \( G \) and \( D \) by solving
\[
\min_{S} f(G, D, S), \quad \text{s.t. } C,
\]  
where we use \( S \) to denote a model and resource dependent representation of the synchronization
strategy, and \( C \) as a set of constraints (developed in §8.3.2). Approaching this problem analyti-
cally needs continuous characterizations of \( S \) and \( f \), which are unavailable. In light of the recent
advance in AutoML [17, 159, 174], we define a domain-specific space considering multiple
synchronization-affecting factors, and resort to search-based methods to find a near-optimal
\( S^* \).
8.2.3 Search Space

When constructing the search space, we have the following considerations. First, instead of optimizing a single factor in a piecemeal fashion as commonly done in existing systems, we seek a unified space covering multiple synchronization-affecting factors, to capture the subtleties between them and their dynamics with different $G$ and $D$ via co-optimization. On the other hand, we want to establish direct correspondences between $S$ and each participating variable of $G$ in synchronization, so that the strategy can adapt with specific variable-wise mathematical properties.

Following the synchronization strategy representation proposed in §7.3, we decompose existing fixed-formed systems or strategies into following orthogonal factors:

- **Variable partitioning**, represented as $p_i = [p_{ij}]_{j=1}^{k_i}$ for the variable $v_i$, where $k_i$ is the number of tensor dimensions of $v_i$, and $p_{ij}$ represents the partition degree on the $j$th axis.

- **Variable placement**, defined as $\{d_i\}_{i=1}^{m} \subset V_D$ which is the set of devices the variable resides. The placement being a single device means $v_i$ is shared across all devices.

- **Synchronization architecture**: we define two types of architecture primitives, namely parameter server (PS) and collective communication (CC), and their architecture-specific semantics. In PS, we use reduction hierarchy to indicate whether parameters are transmitted hierarchically (e.g. from a central CPU to multiple GPUs co-located on the same machine). In CC, we define merge group, where communication primitives assigned with the same group are merged and communicated via a single message, and device order, specifying the message passing order across devices (e.g. tree, ring).

- **Message encoding and decoding**, notated as $c_i$ for $v_i$, introduces compression or decompression schemes to represent how messages are processed before (after) communication, enabling optimizations [92, 156, 169] that exploit structures (e.g. low-rank) exhibited in the messages to reduce message size and fasten network transfer.

Since we focus on synchronous training and optimizing system throughput, we exclude optimizations beyond data-parallel training (e.g. operation partitioning in model-parallel training), or introduce deviation of parameter updates (e.g. staleness which is included in §7.3).

Based on these factors, we express the strategy as $S = \{s_i\}_{i=1}^{V}$ where $s_i$, as a sub-strategy, includes the discrete choices of above factors for each $v_i \in \mathcal{G}$. Note that we decide whether to partition variables or not before any other aspects, so a sub-strategy needs to be generated for each variable partition. The multiple factors span a combinatorial space whose size grows with the size of $G$ and $D$.

We next develop the learning to synchronize framework to approximate the optimal strategy $S^*$. 

133
8.3 **AutoSync: Learning to Synchronize**

Despite the large space, solving Equation 8.1 poses an additional challenge: searching for $S^*$ requires evaluating $f$ for each strategy proposal, which involves distributed execution on clusters and is prohibitively expensive. To make the search tractable, we present the learning to synchronize framework, illustrated in Figure 8.1 and Algorithm ??, with two novel components: runtime simulation of arbitrary $S, G, D$ (§8.3.1), and knowledge-guided strategy search (§8.3.2).

In detail, to reduce real execution, we develop a *domain adaptive simulator* to estimate $f$. The estimation is made possible (without training data) by first designing features agnostic to $G, D$. The features describe critical impacting factors on the runtime using predefined modeling, and can be generalized to any unseen $G$ and $D$. Then, we enhance them using ML models and various raw features extracted from specific $G, D$. In order to direct the search to the subspace where good strategies may locate, we instantiate constraints $C$ in Eq. 8.1 using prior knowledge on synchronization systems.

The final framework, named as *AutoSync*, approaches $S^*$ alternately. In the initial phase, the simulator uses the training-free features to propose the initial batch of strategy candidates, which are executed on real clusters to obtain ground truth runtime. The low-shot data are feedbacked to train the simulator to adapt to $G, D$, achieving improved capability in differentiating high-performing strategies. The process alternates until the optimization budget for real distributed cluster evaluation is exhausted. We next describe the design of the simulator and the guided search.

### 8.3.1 Domain Adaptive Simulator

The simulator takes $(G, D, S)$ as input, and estimates its per-iteration runtime (equivalent with throughput). Because variable partitioning will alter $V_{G,\theta}$, we first infer the new set of variables $V'_{G,\theta}$ based on $G, S$, and let the simulator work with each variable $v_i \in V'_{G,\theta}$, which contains variable shards after partitioning original variables. We define a particular $(G, D)$ target as a *domain*. When equipped with domain-agnostic features, the simulator is capable of estimating the runtime of any $(G, D)$ without training. This is realized by systematic modeling of the runtime in distributed execution.

**Predefined Modeling**

We model the per-iteration runtime $T$ of parallelizing $G$ on $D$ using two contributing components: computation time $T_{\text{comp}}$, and parameter synchronization time $T_{\text{sync}}$. Since: (1) many runtime systems (e.g. TensorFlow [3] or PyTorch [38]) introduce scheduling or parallelization between communication and computation, in practice, there are significant overlaps between the two components; (2) in data-parallel training, it is commonly observed that one component usually dominates the other [55], we simply obtain $T$ via

$$T = \max(T_{\text{comp}}, T_{\text{sync}}).$$

We factorize $T_{\text{comp}}$ and $T_{\text{sync}}$ w.r.t. variables, similar to $S$, and estimate $T_{\text{comp}}(v_i), T_{\text{sync}}(v_i)$ for each $v_i$ based on its $s_i$ in $S$. $T_{\text{comp}}(v_i)$ can be approximated by profiling its corresponding oper-
ation on a single-device. To calculate $T_{\text{sync}}(v_i)$, we split $V'_{G,\theta}$ into variables using PS as $V^{PS}$ and using collective communication as $V^{CC}$, and derive two analytic forms of $T_{\text{sync}}(v_i)$.

**Modeling $v_i \in V^{PS}$.** Synchronizing $v_i$ via PS ($v_i \in V^{PS}$) involves: (1) workers send gradients to servers, (2) servers update parameters, (3) servers send the updated parameters to workers, where (2) is negotiable and (1)(3) are symmetric processes and cost the same amount of time. We denote the original size (e.g. byte size) of the gradient of the variable $v_i$ as $m_i$ (note $m_i$ takes into consideration the sparsity of the gradients when applicable), and assume it will apply the encoding/decoding scheme $c_i \in s_i$, so the actual size of the message (related to $v_i$) to be transferred across devices is $c_i(m_i)$, where we also use $c_i$ to denote the compression function that reduces the original size $m_i$ to $c_i(m_i)$.

Let $w_i$ denote the number of workers involved in synchronizing $v_i$. The parameter transfer process involves transferring data between the GPU device memory and the host memory (RAM) within the same machine, and between the host memory across machines. The first process introduces GPU kernel latency and device-host communication. The second process introduces network overhead (e.g. latency) and network communication. Hence, the communication time $T_{\text{server}}^{PS}$ on the server hosting $v_i$, indexed as $j$, is

$$T_{\text{server}}^{PS}(v_i) = \sum_{k=1}^{w_i} \frac{c_i(m_i)}{b_{j,k}} \cdot r_{i,k}^{p} + \sum_{k=1}^{w_i} \frac{\Pi_d(j, k) \cdot r_{i,k}^{p} \cdot \phi + \delta}{\|V^{PS}\|},$$

where $r_{i,k}$ is the number of replicas of $G$ on worker $k$ if the worker $k$ has multiple GPUs that host multiple replicas of $G$ respectively; $\Pi_d(j, k)$ and $\Pi_p$ are true when server $j$ and worker $k$ locate on different machines and when hierarchical reduction is used, respectively. To interpret Equation 8.2, the first term corresponds to sending messages from each worker $k$ to the server $j$ (and vise versa). The second term captures network overheads that scale linearly with the number of workers, or with the number of replicas when $\Pi_p$ is false. The third term captures the constant GPU memcpy latency.

In addition to the formula, we can construct domain-agnostic features of $v_i$ as:

$$z_i^{PS} = \text{[network transfer, coefficient of } \phi, \text{coefficient of } \delta].$$

As synchronizing any $v_i$ between any pair of nodes can happen simultaneously and is upper bounded by the multi-flow bandwidth, the communication bottleneck may be caused by the slowest transmission, or the total time of transmissions. Thus, we define global features for estimating $T_{\text{sync}}^{PS}$ as:

$$z^{PS} = \max\{z_i^{PS} \text{ for } v_i \in V^{PS}\} \oplus \sum\{z_i^{PS} \text{ for } v_i \in V^{PS}\}/|V^{PS}|,$$

where $\oplus$ is vector concatenation, and $\max, \sum$ are elementwise.

**Modeling $v_i \in V^{CC}$.** For $v_i \in V^{CC}$, we model 5 mostly used collective primitives: AllReduce, ReduceScatter, AllGather, Broadcast and Reduce [107]. Take an example when there are $w$ workers and the device order in the substrategy $s_i$ is a ring [126]. Each primitive sends and
receives \( \frac{2(w-1)}{w}, \frac{w-1}{w}, \frac{w-1}{w}, 1, 1 \) times, respectively, in its applicable scenario for parameter synchronization (e.g. \texttt{AllReduce} for dense gradients or \texttt{AllGather} for sparse gradients [126]). Therefore, we can obtain \( T_{\text{sync}}^{\text{CC}}(v_i) \) using the following formula:

\[
T_{\text{sync}}^{\text{CC}}(v_i) = \Pi_1 \frac{2(w_i - 1)c_i(m_i)}{w_i b_m} + \Pi_2 \frac{(w_i - 1)c_i(m_i)}{w_i b_m} + \Pi_3 \frac{c_i(m_i)}{b_m} + w_i \cdot \phi + \delta,
\] (8.3)

where \( b_m = \min_{(k_1,k_2) \in \text{ring}} b_{k_1,k_2} \) denotes the lowest bandwidth between devices in the ring, since the throughput of a ring is restricted by the lowest bandwidth in the network [148]. \( \Pi_1, \Pi_2, \Pi_3 \) are true when \texttt{AllReduce}, \texttt{ReduceScatter} and \texttt{AllGather}, \texttt{Broadcast} and \texttt{Reduce} are activated, respectively. The formula is derived based on counting how many times each message (i.e. gradients) needs to be passes across the ring, taking into considerations both network transfer overhead as well as the device-host memory swap latency. The total synchronization time for variables assigned with collective communication is \( T_{\text{sync}}^{\text{CC}}(V^{\text{CC}}) = \sum_{i=1}^{|V^{\text{CC}}|} T_{\text{sync}}^{\text{CC}}(v_i) \).

In a similar way with \( v_i \in V^{\text{PS}} \), the domain-agnostic features for \( v_i \in V^{\text{CC}} \) are:

\[
z_i^{\text{CC}} = [\text{network transfer, coefficient of } \phi, \text{coefficient of } \delta]\]

from Eq. 8.3, and the global features of \( T_{\text{sync}}^{\text{CC}} \) are:

\[
z^{\text{CC}} = \max\{z_i^{\text{CC}} \text{ for } v_i \in V^{\text{CC}}\} \oplus \sum\{z_i^{\text{CC}} \text{ for } v_i \in V^{\text{CC}}\}/|V^{\text{CC}}|.
\]

Concatenating \( z^{\text{PS}} \) and \( z^{\text{CC}} \) obtains the set of domain-agnostic features \( z^{\text{pre}} \) for \((G,D,S)\). We can either use the estimated \( T \) to rank different \( S \), or use the constructed features as inputs to ML models, which we elaborate next.

**Domain-specific Modeling**

Once trial data are acquired, we augment \( z^{\text{pre}} \) using raw features \( z^{\text{raw}} \) extracted from \( G,D,S \), and train ML models so to capture their domain-specific characteristics. For each \( v_i, z_i^{\text{raw}} \) vectorizes attributes including variable placement, synchronization architecture, encoding/decoding type, and merge group from \( S \), bandwidth and the number of replica devices of each node from \( D \), and variable size, dimensions, the sparsity of gradients, data types, and information of partitioned shards from \( G \). Combining \( z^{\text{pre}} \) and variable-specific raw features \( z_i^{\text{raw}} \), we adopt three different ML models: (1) a linear model, (2) a recurrent neural network (RNN), and (3) a graph attention networks (GAT) [170], to learn from trial data and make more accurate predictions of \( f \).

**Linear model.** The linear model, simply written as \( \hat{f} = z^{\text{pre}} \cdot \theta \), introduces trainable weights \( \theta \) and predicts the runtime \( \hat{f} \) using only global features \( z^{\text{pre}} \).

**RNN.** We use a RNN to model different \( G \) with varying number of variables, so as to inject \( z_i^{\text{raw}} \). The RNN first concatenates \( z_i^{\text{pre}} \) with \( z_i^{\text{raw}} \), and then transforms the combined features via an MLP. The results are aggregated into \( |V_{G,\theta}| \) features corresponding to the variables in the \( V_{G,\theta} \).
A bidirectional LSTM scans them following the forward-backward propagation order preserved in \( G \). At last, the prediction \( \hat{f} \) is obtained via an extra MLP (Figure 8.1). Detailed formulas are below:

\[
\begin{align*}
    z_i &= \text{MLP}_i \left( z_i^{\text{raw}} \oplus z_i^{\text{pre}} \right), \\
    h_{i+1} &= \text{LSTM} \left( h_i, z_i \right), \\
    h'_{i+1} &= \text{LSTM} \left( h'_i, z_{\left| V'_G, \theta \right|i+1-i} \right), \\
    \hat{f} &= \text{MLP}_o \left( \left| h_{\left| V'_G, \theta \right|i}, h'_{\left| V'_G, \theta \right|i} \right| \right).
\end{align*}
\]

**GAT.** We bring in GAT to model the raw graph structure of \( G \). To do so, we prune \( G \) into a graph \( G_p = (V_{G_p}, E_{G_p}) \) with only variable (including partitioned) nodes and edges connecting them. For each node, similar to RNN, its raw and predefined features are concatenated, and features of all nodes and corresponding edge information are fed to a GAT for encoding graph structure as:

\[
\{ z_i \} = \text{GAT}(\{ z_i^{\text{raw}} \oplus z_i^{\text{pre}} \text{ for } v_i \in V_{G_p}, E_{G_p} \}),
\]

shown in Figure 8.1. The node features are then aggregated into graph-level ones, followed by an MLP to get the runtime estimation \( \hat{f} \), following formulas:

\[
\begin{align*}
    \{ z_i \}_{i=1}^{\left| V'_G, \theta \right|} &= \text{GAT}(\{ z_i^{\text{raw}} \oplus z_i^{\text{pre}} \text{ for } v_i \in V_{G_p}, E_{G_p} \}), \\
    z &= \frac{1}{\left| V'_G, \theta \right|} \sum_{i=1}^{\left| V'_G, \theta \right|} (\{ z_i \}_{i=1}^{\left| V'_G, \theta \right|}), \\
    \hat{f} &= q(z),
\end{align*}
\]

**Training Objective**

Accurately predicting runtime can be challenging due to uncertain factors on a distributed cluster, we instead train the simulators with the pair-wise logistic ranking loss [9, 17]:

\[
\sum_{i=1}^{n} \sum_{j=1}^{n} \mathbb{I}(f_i + \sigma \mathbb{I}_1(P_i, P_j) > f_j) \log(1 + \exp(\hat{f}_j - \hat{f}_i)),
\]

where \( f_i, f_j \) are ground truth runtime, \( \mathbb{I} \) is an indicator function, and \( n \) denotes the number of training examples. Note that we augment the original ranking loss with a penalty term \( \sigma \mathbb{I}_1(P_i, P_j) \), where \( \sigma \) is a non-negative threshold, \( P_i \) is the total number of partitions in \( S_i \), and \( \mathbb{I}_1(x, y) \) outputs 1 when \( x > y \), -1 when \( x < y \), and 0 otherwise. This term additionally considers inherent partitioning overheads, and alleviates the bias toward heavily partitioning variables.

**8.3.2 Knowledge-guided Search**

The search uses the simulator to evaluate a strategy, and proposes candidates with low predicted runtime \( \hat{f} \). It then selects a small number of qualified candidates from a large set of proposals for trial execution, as shown in Figure 8.1. We implement two search algorithm variants: random search and genetic algorithm (GA) [29]. The detailed search algorithm can be found in the supplementary.
Knowledge Constraints

As the strategy space is exponentially large, it is inefficient to grid search the entire space. We instantiate the constraints $C$ in Eq. 8.1 using two system design principles to restrict the exploration within promising regions.

**Load balancing constraint $c_{lb}$**: which is vital to alleviate the communication bottleneck. When deciding the placement for each $v_i \in V^{PS}$, $c_{lb}$ enforces to sample the placement from a multinomial distribution over all participating nodes in $D$, where each node’s probability of being chosen correlates to its current communication load and maximum bandwidth, so that nodes with higher available bandwidth are more likely to be sampled. This allows generating randomized solutions while approximately maintaining a balanced status across all nodes.

**Adjacent merging constraint $c_{am}$**: the fusion of collection operations should correspond to the model forward-backward propagation order in $G$, as merging two operations in the head and tail of the model would prevent low-level scheduling overlapping communication and computation. $c_{am}$ is introduced to ensure variables adjacent to each other in $G$ are more likely to be grouped together. We show the $c_{lb}$ and $c_{am}$ empirically improve search efficiency and quality in §8.4.2.

Select Evaluation Candidates

Similar to AutoTVM [17], we select the final set of candidates, that have minimized weighted sum of $\hat{f}$ and internal similarity, for distributed execution.

Formally, we solve the following optimization problem:

$$\min_{\{S_i\}_{i=1}^{K}} \sum_{i=1}^{K} \hat{f}(S_i) + \alpha \sum_{i=1}^{K} \sum_{j=1}^{K} \text{sim}(S_i, S_j),$$  \hspace{1cm} (8.4)

where $\{S_i\}_{i=1}^{M}$ are the $M$ qualified strategies filtered first using the simulator score, $\alpha$ denotes a trade-off coefficient, and sim is a pairwise similarity function between strategies. Solving Eq. 8.4 helps deliver a set of candidates that trade off between low predicted runtime (exploitation) and low similarity (exploration). The above problem is a typical problem of submodular minimization, and we resort to the greedy algorithm for an approximate solution [79].

Estimating the Similarity between Strategies

Since we do not have a continuous representation of strategies, we have developed two approaches to estimate the similarities between strategies. The first approach estimates the similarity of the two strategies $S_1, S_2$ by comparing each sub strategy $s_{i1} \in S_1$ and $s_{i2} \in S_1$ corresponding to the variable $v_i$, and counting how many choices of each synchronization-affecting factor are the same, and use the final count as a measure of their similarity (higher is more similar). The second approach takes the cosine similarity between the hidden outputs of the simulator, whose inputs are $S_1$ and $S_2$, as a proxy of the strategy similarity. Empirically, we find the two approaches perform similarly, but the first similarity function does not depend on the simulator thus can be used when no trained simulator is available (e.g. for the baseline $\text{AutoSync}(-s)$).

138
Figure 8.1: **Left**: Learning to synchronize framework. Initially, the simulator utilizes domain-agnostic features (§8.3.1) to explicitly estimate the runtime so to select promising strategies for evaluation (the blue line). After trials, the real runtime data are feedbacked to train the ML-based simulator to adapt to specific $G, D$, enhancing its capability in differentiating high-quality strategies. Gradually, the ML-based simulator takes over and directs the search (the red line). **Right**: Illustrations of the RNN and GAT simulators.
8.3.3 Low-cost Optimization using Transfer Learning

Performing end-to-end strategy optimization from scratch for a target domain $G_t, D_t$ on large clusters or large models is costly, especially if $G_t$ will be only trained once (one-shot training). For unseen $G_t, D_t$, we consider transferring simulators trained on the data from a source domain $G_s, D_s$. This would be advantageous if the source domain data already exist, or any of $G_s$ and $D_s$ is small so their trial run data is cheap to collect. The transferability is made possible by our feature design – the predefined features are universal due to its domain-agnostic nature. The per-variable raw features are extracted from $G, S, D$, where $G$ is the generic low-level dataflow graph representation, capable of expressing all NN models; $S$, based on $D$, is generated from a strategy space invariant to models as well. Hence, the feature components of different $(G, D, S)_i$ are invariant, leaving the only variability as the the length of features, caused by the different number of variables in different $G$, which, however, is absorbed by models like GAT and RNN that operate on inputs with variable lengths. §8.4.3 validates the effectiveness of the transferable feature representation.

8.4 Evaluation

We evaluate AutoSync’s components and end-to-end performance in §8.4.2. We investigate transferability of trained simulators across domains in §8.4.3.

8.4.1 Setup

Models

We generate strategies on top of AutoDist, and resort to distributed TensorFlow for distributed execution [43]. We treat $f$ as system performance, and conduct synchronous training$^1$ on 10 models with standard settings suggested by MLPerf [97], including an enlarged dense (x16x32) version of the neural collaborative filtering (NCF) [59], Transformers [145] and BERT variants [32], and various CNNs [57, 63, 129], listed in Table 7.3. We managed to train all models to the suggested accuracy, hence skip the comparisons on convergence.

Clusters

We conduct experiments on two clusters $(D)$:

1. *Cluster A* is an in-house cluster with 11 nodes, each equipped with a TITAN X GPU and 40GbE Ethernet switch;

2. *Cluster B* is based on AWS, consists of 4x g4dn.12xlarge nodes, each with 4 NVIDIA T4 GPUs and 50GbE full bandwidth.

They correspond to the cluster A2 and B7 in Table 7.2.

$^1$In this paper, we do not consider synchronization-affecting factors that would alter the algorithm or result as in the original single-node code, such as lossy compression [92], staleness [60], etc.
Hand-optimized Baselines

We introduce two strong hand-optimized synchronization systems as external baselines.

- **Horovod** [126] is one of the most adopted open source synchronization systems for data-parallel distributed ML. It uses *AllReduce (AllGather)* to synchronize dense (sparse) gradients of all model variables, and utilizes BO to autotune the merge scheme for multiple collective operation based on collected trial data in warm-up runs\(^2\). Per our experiments on our cluster setup, it reports up to 2x speedup than a Google-provided parameter server implementation based on distributed TensorFlow as runtime\(^3\). Horovod uses NCCL 2.4.7 for collective communication, which is the same with AutoSync.

- **PS** is a highly tuned parameter server implementation with multiple optimizations from recent PS literature incorporated. We elaborate a few notable optimizations: (1) it maintains load balance by partitioning large variables and evenly placing the shards across servers depending on their available bandwidth (i.e. correlates to its maximum bandwidth and the current load as a parameter server); (2) it uses the BO algorithm to decide the partitioning size following ByteScheduler [115]; (3) it communicates sparse gradients (*IndexSlices* in TensorFlow) using *Gather* and *Scatter* primitives (instead of *Reduce* and Broadcast) \(^76\) to reduce communication overhead; (4) it performs hierarchical Reduce (or Gather) and Broadcast (or Scatter) on nodes with more than 1 GPUs to reduce network traffic. The optimizations are implemented on top of TensorFlow 2.0 as well, so the backend also relies on distributed TensorFlow 2.0 for distributed execution, same with AutoSync.

We use them as collective- and PS-based baselines, respectively. Note that both methods use the same TF and NCCL version for distributed execution and communication with AutoSync, preventing variations caused by systems.

Public Datasets

Throughout our research, we have collected a dataset containing nearly 10k data points of \(\{(G, D, S), f\}\), where \(G\) is one of the DL models in Table 7.3, \(D\) is one of the cluster setups from Table 7.2, \(S\) is randomly sampled strategy from the proposed strategy space, and \(f\) is the groundtruth runtime collected via real distributed execution.

The dataset contains strategies sampled for all 11 models in Table 7.3, ranging from fixed-formed strategies such as those used in specialized systems, and randomly explored strategies by AutoSync during the strategy auto-optimization. The data are organized into multiple folders where each folder corresponds to a domain of \((G, D)\). Hence the dataset used in the main paper is subset containing several domains as parts of the whole collected dataset. For quick access, we have provided scripts that read \(G\) as dataflow graphs in standard TensorFlow 2.0 format, and read the strategies and runtime into json formats.

\(^2\)Details of the optimization can be found at [https://github.com/horovod/horovod/blob/master/docs/autotune.rst](https://github.com/horovod/horovod/blob/master/docs/autotune.rst).
\(^3\)Details at [https://github.com/tensorflow/examples/blob/master/community/en/docs/deploy/distributed.md](https://github.com/tensorflow/examples/blob/master/community/en/docs/deploy/distributed.md), on multiple CNNs such as ResNet101 and InceptionV3.
Table 8.1: Comparisons of different model instantiations of the simulator.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Linear</th>
<th>RNN</th>
<th>GAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCF-dense, A</td>
<td>0.771</td>
<td><strong>0.894</strong></td>
<td>0.810</td>
</tr>
<tr>
<td>NCF-dense, B</td>
<td>0.826</td>
<td><strong>0.913</strong></td>
<td>0.830</td>
</tr>
<tr>
<td>VGG-16, A</td>
<td><strong>0.868</strong></td>
<td>0.796</td>
<td>0.753</td>
</tr>
<tr>
<td>VGG-16, B</td>
<td>0.833</td>
<td><strong>0.839</strong></td>
<td>0.692</td>
</tr>
<tr>
<td>BERT-base, A</td>
<td>0.758</td>
<td>0.746</td>
<td><strong>0.775</strong></td>
</tr>
<tr>
<td>BERT-base, B</td>
<td>0.807</td>
<td><strong>0.867</strong></td>
<td>0.760</td>
</tr>
<tr>
<td>BERT-large, A</td>
<td>0.780</td>
<td><strong>0.847</strong></td>
<td>0.771</td>
</tr>
<tr>
<td>BERT-large, B</td>
<td><strong>0.796</strong></td>
<td>0.755</td>
<td>0.784</td>
</tr>
</tbody>
</table>

Table 8.2: The per-iteration time statistics of 200 strategies found by RS and GA on (VGG16, A).

<table>
<thead>
<tr>
<th>stats</th>
<th>RS</th>
<th>GA</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean (s)</td>
<td>1.07</td>
<td>0.67</td>
</tr>
<tr>
<td>std (s)</td>
<td>0.63</td>
<td>0.03</td>
</tr>
<tr>
<td>min (s)</td>
<td><strong>0.60</strong></td>
<td>0.64</td>
</tr>
<tr>
<td>max (s)</td>
<td>2.34</td>
<td>0.79</td>
</tr>
</tbody>
</table>

8.4.2 End-to-end Results and Ablation Studies

Comparing Model Instantiations

To compare the linear model, RNN and GAT, we construct datasets using trial data collected on 6 different settings, and train them as simulators, respectively. We report their ranking accuracy on held-out test sets in Table 8.1. RNN, by leveraging raw features, outperforms linear model mostly. GAT, though additionally modeling the graph structure of $G$, does not demonstrate substantial advantages. We hypothesize that GAT might need more data for training, which are unavailable in our budgeted search. We hence use RNN by default in the rest of the chapter.

Search Algorithm Comparisons

We implement both random search and genetic algorithm (GA) [29] for searching. Table 8.2 right compares them regarding optimizing the strategy for VGG16. While GA finds strategies with better average quality, it constantly gets stuck at the local minima if working with an untrustworthy simulator $\hat{f}$ (instead of using real execution data $f$). Hence in our experiments, we by default use random search.

Auto-optimization Results

We use AutoSync to optimize the strategies of NCF-dense (122M), VGG16 (138M), and BERT-large (340M). They cover 3 different NN families but are all considered “difficult-to-parallelize”
workloads because of having $>100$M parameters. We compare two AutoSync variants with external baselines:

1. AutoSync(-s) where the simulator is disabled for searching. It randomly explores 30K strategies and selects 200 candidates that have minimized similarity ($\S8.3.2$).

2. AutoSync: the full AutoSync with the budget of real evaluation on clusters as 200.

To obtain the runtime $f$, we run 10 warm-up iterations, then another 40 iterations of training, whose runtime is averaged as the groundtruth.

Figure 8.2 compares the best found strategy in 200 trials by two variants with the two manually optimized baselines. In 4 out of 6 settings, AutoSync(-s), without a simulator, discovers strategies up to 1.4x faster than the best one in PS and Horovod. With the simulator, AutoSync finds strategies 1.2x to 1.6x faster than baselines. To interpret the speedup, practically BERT-large needs 2M steps [99] of training to its reported accuracy with batch size 128 (batch size 8 on 16 GPUs). A 1.2x speedup reduces the training time by 7 days, and saves approximately $2200$ AWS credits per training job on Cluster B. Moreover, in practice a model needs to be retrained when being applied to unseen data, but a trained simulator can be repeatedly used across jobs. Comparing AutoSync to AutoSync(-s): besides higher quality, the simulator guides the search to solutions sooner, e.g. on (BERT-large, A), AutoSync locates strategies 1.25x faster than PS in about 50 trials, and 1.2x using only 30 trials on Cluster B. Similar patterns are observed in other settings as well.

Search Space Evaluation

We define the metric hit rate – the percentage of strategies found better than hand-optimized baselines during explorations, illustrated in the right axis of Figure 8.2. Except on (NCF-dense, B), AutoSync(-s) reports positive hit rates and the rate is considerably large on (VGG16, B) (42.0%) and (BERT-large, B) (28.5%). When augmented with a simulator, AutoSync frequently hits better strategies, especially on more complex models VGG16 and BERT-large (> 70%). This verifies that factorizing the strategy w.r.t. each variable and co-optimizing multiple factors form a promising space with a profound set of strategies better than manually optimized.

Knowledge Constraints

Figure 8.2 contrasts another variant, AutoSync(-s,-k), where the knowledge constraints in $\S8.3.2$ are removed from AutoSync(-s). AutoSync(-s,-k) mostly visits strategies below hand-optimized baselines, especially on complex models with a larger search space – on BERT-large all strategies by AutoSync(-s,-k) are far below baselines hence are skipped in the plots. Incorporating system design principles as knowledge is key to make the search manageable.
### Feature Importance

We ablate $z^{raw}$ and $z^{pre}$ and reveal their individual effect in Table 8.3. Specifically, we train RNN simulators with: (1) only predefined features $z^{pre}$, (2) only raw features $z^{raw}$, (3) the full features, under 6 $(G, D)$ settings, and compare their ranking accuracy on test sets. Using full features achieves the best accuracy, demonstrating that the predefined and raw features may contain complementary information. On the other hand, the predefined features can be beneficial at the initial phase of search as it is possible to directly rank strategies using them without training.

<table>
<thead>
<tr>
<th>Setting</th>
<th>full</th>
<th>raw only</th>
<th>predefined only</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCF-dense, A</td>
<td>0.894</td>
<td>0.894</td>
<td>0.883</td>
</tr>
<tr>
<td>NCF-dense, B</td>
<td>0.913</td>
<td>0.907</td>
<td>0.873</td>
</tr>
<tr>
<td>VGG16, A</td>
<td>0.796</td>
<td>0.785</td>
<td>0.734</td>
</tr>
<tr>
<td>VGG16, B</td>
<td>0.839</td>
<td>0.837</td>
<td>0.816</td>
</tr>
<tr>
<td>BERT-large, A</td>
<td>0.847</td>
<td>0.848</td>
<td>0.850</td>
</tr>
<tr>
<td>BERT-large, B</td>
<td>0.755</td>
<td>0.746</td>
<td>0.735</td>
</tr>
</tbody>
</table>

Table 8.3: Studies on feature importance (pairwise ranking accuracy is reported).
Figure 8.2: Comparing AutoSync, AutoSync (-s), AutoSync (-s, -k) on (left axis) the improvement (higher is better) of the best found strategy over baseline, (right axis) the percentage of strategies better than baseline (higher is better), w.r.t. the number of trials conducted in 200 trials. The baseline (1x) is the better one of PS and Horovod). The average over 3 runs is reported. A curve is skipped from the plot if it is too below the baseline.
8.4.3 Transferring Trained Simulators

Transferability Studies

Different from §8.4.2, we train RNN simulators using trial data from a source domain $G_s, D_s$ to
rank the strategies in unseen target domains $G_t, D_t$, and report the ranking accuracy in Table 8.4
(more in supplementary). In general, we note that: (1) Models with similar architectures exhibit
higher transferability. With fixed $D$, transformer [145] based models transfer between each other
pretty well, with the lowest accuracy at 0.76; the transferability is slightly compromised when
$D$ is changed, due to increased domain distance. This hints we can use a simulator trained for
smaller BERT models to optimize the strategy of a larger BERT model. (2) When $G$ is fixed,
transferability is observed across $D$. In practice, we might pretrain a simulator using in-house
cheap clusters (e.g. cluster A), and deploy the simulator for training jobs on more expensive and
larger-scale clusters.

<table>
<thead>
<tr>
<th>Source → Target</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>(BERT-3L, A) → (BERT-3L, B)</td>
<td>0.8672</td>
</tr>
<tr>
<td>(Transformer, A) → (BERT-base, A)</td>
<td>0.7674</td>
</tr>
<tr>
<td>(BERT-3L, A) → (BERT-base, A)</td>
<td>0.7591</td>
</tr>
<tr>
<td>(BERT-3L, B) → (BERT-base, B)</td>
<td>0.8100</td>
</tr>
<tr>
<td>(NCF-dense, A) → (BERT-base, B)</td>
<td>0.7904</td>
</tr>
<tr>
<td>(VGG16, A) → (BERT-base, B)</td>
<td>0.7298</td>
</tr>
<tr>
<td>(BERT-3L, A) → (BERT-base, B)</td>
<td>0.6992</td>
</tr>
<tr>
<td>(BERT-base, A) → (BERT-base, B)</td>
<td>0.6852</td>
</tr>
<tr>
<td>(VGG16, B) → (BERT-base, B)</td>
<td>0.6774</td>
</tr>
<tr>
<td>(Transformer, A) → (BERT-base, B)</td>
<td>0.6672</td>
</tr>
<tr>
<td>(BERT-3L, A) → (Transformer, A)</td>
<td>0.8866</td>
</tr>
<tr>
<td>(Transformer, B) → (Transformer, A)</td>
<td>0.8305</td>
</tr>
<tr>
<td>(Transformer, A) → (Transformer, B)</td>
<td>0.8171</td>
</tr>
<tr>
<td>(BERT-3L, A) → (Transformer, B)</td>
<td>0.808</td>
</tr>
<tr>
<td>(NCF-dense, A) → (NCF-dense, B)</td>
<td>0.7694</td>
</tr>
<tr>
<td>(VGG16, A) → (VGG16, B)</td>
<td>0.7966</td>
</tr>
<tr>
<td>(BERT-3L, A) → (VGG16, B)</td>
<td>0.5583</td>
</tr>
<tr>
<td>(ResNet50, A) → (ResNet101, A)</td>
<td>0.6057</td>
</tr>
<tr>
<td>(VGG16, A) → (ResNet101, A)</td>
<td>0.5600</td>
</tr>
<tr>
<td>(VGG16, A) → (ResNet50, A)</td>
<td>0.7156</td>
</tr>
<tr>
<td>(VGG16, A) → (DenseNet121, A)</td>
<td>0.7596</td>
</tr>
<tr>
<td>(ResNet101, A) → (ResNet101, B)</td>
<td>0.7187</td>
</tr>
<tr>
<td>(DenseNet121, A) → (ResNet50, B)</td>
<td>0.5266</td>
</tr>
<tr>
<td>(VGG16, A) → (InceptionV3, A)</td>
<td>0.7857</td>
</tr>
</tbody>
</table>

Table 8.4: The target domain test accuracy under different transfer learning settings.
End-to-end Results on Transfer Learning

We now transfer trained simulators to guide end-to-end optimizations. We deliberately target difficult-to-parallelize models and expensive clusters: (BERT-base, A), (BERT-base, B), and (VGG16, B). We set a smaller budget of 100 trials, and do not use any data from target domains for model selection or finetuning. Figure 8.3 illustrates the optimization progress. On (VGG16, B), surprisingly, a well-trained simulator from (VGG16, A) can find strategies 1.5x faster as soon as in 5 trials. On (BERT-base, B), we experiment with 3 source domains and notice that the domain distance does impact the end-to-end results: source (BERT-3L, B) achieves better quality and efficiency than (BERT-base, A), both better than (BERT-3L, A); (BERT-3L, A) has the largest domain distance from the target. Overall, we manage to transfer simulator trained on “cheap” domains to find good strategies in only a few trials in “expensive” domains. Besides the reduction on the number of trials, transferring a simulator effectively decreases the wall-clock time taken in auto-optimization as it bypasses simulator training, which is advantageous for scenarios where single-shot model training happens often.
Figure 8.3: Transferring trained simulators from different source domains to 3 target domains, compared to untransferred AutoSync and AutoSync with a budget of 100 trials. The average of three runs is reported.
8.5 Additional Related Work

ML for Systems

There is a surge of interest in applying ML to solve system problems. Mirhoseini et al. [100, 101] develops reinforcement learning (RL) frameworks to decide the placement of nodes in dataflow graphs. Paliwal et al. [112] combines RL and genetic algorithms (GA) to minimize the execution cost of NN graphs for compilers. AutoTVM [16] builds an ML-based pipeline to generate operator implementations better than hand-designed. AutoSync belongs to this line of work: it uses ML to optimize data-parallel synchronization and addresses the unique challenges therein.

Synchronization System Autotuning

Many data-parallel ML systems demonstrate certain levels of autotuning capability. Horovod [126] and ByteScheduler [115] introduce adjustable knobs and credit size, respectively, and use Bayesian optimization (BO) [40] to autotune their values. Parallax [76] develops a 3-parameter linear model, learned via trial data, to find the best partitioning for sparse variables in PS. These work focuses on autotuning one or two hyperparameters of a specific strategy; AutoSync contrasts them by co-optimizing a generic and holistic representation of synchronization. Among them, the closest to ours is AutoTVM [17]. We draw insights from AutoTVM, but address a fundamentally different problem – data-parallel auto-distribution of ML training on clusters – which requires the problem-specific design of strategy representations, features, runtime simulations, etc.

Automatic ML Parallelization

Automating the parallelization of ML programs is an ultimate goal in distributed ML. FlexFlow [70, 71] proposes the SOAP representation to express partitioning schemes of NN layers, and an MCMC-based algorithm to search for optimal partitioning configurations. TOFU [149] concerns the partitioning to finer-grained computational operations in DL dataflow graphs. They intersect with AutoSync as data parallelism (which is AutoSync focuses on) can be equivalently represented as partitioning all the layers/operations along their batch dimension. However, beyond partitioning, AutoSync models many other synchronization-affecting factors, such as communicating architectures, sharding, and merging schemes, etc., which differentiates us from this line of work.
Chapter 9

Conclusion and Future Work

9.1 Conclusion

9.2 Future Directions
Bibliography

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, and Michael Isard. Tensorflow: A system for large-scale machine learning. arXiv preprint arXiv:1605.08695, 2016. 3, 3.5, 3.5.1, 2, 3.5.1, 3.6, 3.6, 6.2.1, 6.2.1, 6.4, 6.4.2, 6.5.2, 6.5.2, 6.5.5, 6.5.5, 6.5.5, 6.5.5, 6.6.1, 6.7, 7, 1, 7.4.4, 8.3.1


[19] Trishul Chilimbi, Yutaka Suzue Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In OSDI, 2014. 3.1.1, 3.2.2, 3.3, 3.5, 3.5.4, 3.5.4, ??, 3.5.5, 3.6, 4, 4.1.2, 4.1.2, 4.4.1, 4.4.2, 4.4.2, 5.1, 5.3.3, 5.5, 7.1, 7.2


[21] Adam Coates, Brody Huval, Tao Wang, David J. Wu, Andrew Y. Ng, and Bryan Catanzaro. Deep Learning with COTS HPC Systems. In ICML, 2013. 3.6, 4

[22] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R Ganger, Phillip B Gibbons, et al. Exploiting bounded staleness to speed up big data analytics. In 2014 {USENIX} Annual Technical Conference ({USENIX} {ATC} 14), pages 37–48, 2014. 4.1.2, 4.1.2, 4.2.2, 4.3.2, 4.3.2, 4.4.2, 5.1,
5.2.2, 5.5, 7.1

[23] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In Proceedings of the Eleventh European Conference on Computer Systems, page 4. ACM, 2016. 3, 3.2.2, 3.4.1, 3.4.1, 3.5.1, 3.5.2, 3.5.4, 3.6, 3.6, 4, 5.1, 5.2.2, 5.3.3, 5.5, 6.5.5, 7, 7.3.3, 7.7


[28] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large Scale Distributed Deep Networks. In NIPS, 2012. 3.5.5, 3.6, 4.1.2, 4.4.2, 5.1, 5.2.2, 5.3.3, 5.5


[31] Jia Deng, Nan Ding, Yangqing Jia, Andrea Frome, Kevin Murphy, Samy Bengio, Yuan Li, Hartmut Neven, and Hartwig Adam. Large-scale object classification using label relation graphs. In European conference on computer vision, pages 48–64. Springer, 2014. 4.5


[38] Facebook. http://pytorch.org/. 3, 6.4.2, 6.6.1, 6.7, 8.3.1


[60] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip. B. Gibbons, Garth A. Gibson, Greg R. Ganger, and Eric P. Xing. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *NIPS*, 2013. 3.4.1, 4.1.2, 4.3.1, 5.1, 5.2.2, 5.5, 7.1, 7.1, 7.2, 7.3.2, 1


155


[79] Andreas Krause and Daniel Golovin. Submodular function maximization. 8.3.2


[83] Abhimanu Kumar, Alex Beutel, Qirong Ho, and Eric Xing. Fugue: Slow-worker-agnostic distributed learning for big models on big data. In *Artificial Intelligence and Statistics*, pages 531–539, 2014. 5.2.2

[84] Rasmus Munk Larsen and Tatiana Shpeisman. Tensorflow graph optimizations. 2019. 7.4.3, 7.6.2

[85] Quoc V Le, Rajat Monga, Matthieu Devin, Kai Chen, Greg S Corrado, Jeff Dean, and Andrew Y Ng. Building High-level Features Using Large Scale Unsupervised Learning. In *ICML*, 2012. ??, 3.5.5, 3.5.5, 3.6


[101] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. A hierarchical model for device placement. 2018. 8.5


[105] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and


[111] NVIDIA. http://docs.nvidia.com/cuda/nvrtc/index.html. 6.5.4


[116] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In OSDI, 2010. 4.1.2

[117] Dan Quinlan. Rose: Compiler support for object-oriented frameworks. Parallel Processing Letters, 10(02n03):215–226. 6.5.4


[119] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand,


[126] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018. 3.6, 1, 7.1, 7.1, 7.2, 7.2, 7.6.2, 7.7, 8.1, 8.3.1, 8.4.1, 8.5


160


[152] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Managed Communication and Consistency for Fast Data-parallel Iterative Analytics. In *SoCC*, 2015. 3, 3.1.1, 3.4, 3.4.1, 3.6, 4.3.2, 7.2, 8.1

[153] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *SoCC*, 2015. 5.5

[154] Ren Wu. Deep image: Scaling up image recognition. 4.1.2, 4.6


[163] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Zhiheng Huang, Brian Guenter, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Huaming Wang, et al. An introduction to computational networks and the computational network toolkit. Technical report. 3.5, 3.5.4, 3.5.4, 3.6


163


