Thesis Proposal

Machine Learning Parallelism Could Be Adaptive, Composable and Automated

Hao Zhang

November 19, 2019

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 © 2019 Hao Zhang
Abstract

In recent years, the pace of innovations in the fields of machine learning 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 or GPUs, or even multiple computing nodes over a network (distributed machine learning). As ML and deep learning models become more structurally complex, these systems have struggled to provide excellent all-round performance on a wide variety of models. In this thesis, we propose a simple design principle, adaptive parallelism, to guide the design of algorithms and systems for large-scale distributed ML. Following this principle, we derive a series of new parallelization strategies that interpolate existing ML parallelisms and adapt to the model and cluster specifications. We examine these strategies and show that they boost the scalability and efficiency for one order of magnitude across a diverse set of models, environment, and distributed ML workloads, and open up the space for new model and algorithm design.

Generalized from multiple instantiations of this methodology, we develop ways of expressing various ML parallelisms, originated from different aspects, including synchronization architecture, model partitioning and placement, consistency, to name a few, within a unified representation. We demonstrate that the representation allows for rapid composing of adaptive parallelization strategies for unseen models from existing parallelisms, simplification of parallel ML programming, and leads to system implementations with better abstraction. 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. To overcome this obstacle, we propose to investigate learning-based methods for distribution strategy search over the space spanned by the representation, in order to automate ML parallelization.

1 Introduction

In recent years, the pace of innovations in the fields of machine learning (ML) and artificial intelligence (AI) has accelerated, with noteworthy advances across multiple application areas, for examples, programs that analyze and classify unstructured data such as images, video, audio, human-written text, and graphs. The common denominators that tie these applications together are their use of complex, hierarchically structured, and deep models, and their reliance on large volumes of data to satisfy the training requirements for these complex models. In particular, the largest machine learning models may involve hundreds of millions of parameters [22, 56] if they are dense models, or even billions of parameters [77] in the case of sparse models.

To cope with the sheer computational complexity of training such large models on large datasets, researchers in the systems and ML research communities have created software systems that parallelize training algorithms over multiple CPUs or GPUs (multi-device parallelism) [?], or even multiple computing nodes over a network (distributed parallelism) [14, 40, 71, 78, 80]. In practice, however, notable technical barriers have been observed that prevent a simple adoption of parallel machine learning. They stem from at least three sources: (1) Machine learning models exhibit rich varieties. Achieving parallelization speedups requires parallelization techniques that fit well with the statistical or algorithmic properties of the given model. Hence, parallelizing
machine learning requires domain expertise to map the right parallelisms with (each component of) the model. (2) Distributed parallel code is by nature difficult to write and debug, especially when coupled with ML. (3) In a distributed parallel environment, the costs of communicating and synchronizing results between devices and nodes are highly non-trivial and would result in degenerated performance or scalability if lacking proper systematic optimizations.

In this thesis, we study the applicability of different ML parallelization techniques (which we will call parallelisms as well in the following text). Based on the structured and composable nature of DL models, we develop a methodology that derives parallelization strategies that can adapt to the detailed computational pattern or ML properties of the model or their building block components, and achieve more efficiency and scalability on a variety of models and applications. Generalizing this methodology, we develop a unified representation that is capable of expressing various parallelisms designed for different aspects of parallel ML. The representation enables system design that isolates the sophistication of distributed systems from ML prototyping, simplifies parallel ML programming, and allows for rapidly composing new distribution strategies from a few base ML parallelisms that adapt to unseen models. As a result, we observe improved results in a few ML applications.

2 Thesis Overview

2.1 Thesis Statement

In our finished and proposed work, 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.

2.2 Contributions

This thesis statement is supported with three major research components, with my contributions or proposed work in each part elaborated below:

Adaptive ML 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, execution scheduling, and resource allocation, etc. With the growing complexity of ML/DL models, we observe that parallelizing or distributing ML workloads with one monolithic strategy (e.g. via parameter servers) does not suffice, or sometimes even fails. Instead, we develop a principled methodology called adaptive parallelism – that factors the parallelization strategy with respect to the model. Based on the compositability of ML programs, it interpolates existing parallelisms and 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 research yields the following contributions:

- It introduces an adaptive scheduling algorithm, wait-free backpropagation (WFBP) (section 4.1.1), and a hybrid communication architecture (section 4.2). 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 (section 4.1.1), specialized to address the memory shortage in distributed deep learning on GPU clusters.

- Through both theoretical and empirical studies on how staleness impacts the training of various models and deep learning building blocks (section 4.5), it reveals that the consistency model should also flexibly adapt to model specification to trade-off between system throughput and statistical efficiency.

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

- Finally, this thesis further proposes to explore batchsize-adaptive resource scheduling (section 4.4) – that improves resource allocation on shared clusters by dynamically adapting with model details and training status.

**Representations of adaptive parallelisms.** We generalize from multiple instantiations of adaptive parallelisms presented in this first part – conditioned on the dataflow graph 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 base parallelisms as building blocks; they simplify parallel ML programming for DL models, isolate the sophistication of distributed systems from the ML prototyping process; they lead to a more generic abstraction of the system implementation – that are less specialized for one or two particular models or parallelization strategies (which is the case for most existing distributed ML systems). This part of research contains the following contributions:

- It formalizes a unified representation of distribution strategies for static DL models. This representation allows parallelization strategies to only depend on model and resource specifications, and to be composable from base parallelisms. Based on this representation, we develop Arion, a system for low-barrier distributed ML programming and fast prototyping of distribution strategies for unseen models.

- It contributes to an open source DL framework, DyNet (section 5.2.2), specialized for dynamic neural networks based on dynamic declaration.

- One step further, it introduces a new vertex-centric representation for dynamic neural networks. The representation derives a vertex-centric programming model, 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.

**Search strategies for adaptive ML Parallelisms.** Backed by the developed representations and system implementations, we propose to investigate distribution strategy search that uses learning-based methods to search over the strategy space spanned by the representation and automatically generates the optimal distributed strategy conditioned on the input model.

3 Preliminaries

3.1 Dataflow Graphs and DL Frameworks

Deep learning is distinguished from other machine learning algorithms mainly by its use of deep neural networks, a family of ML models that exhibit strong composability – they are usually built by many interconnected layers, each composed of various mathematical operations (e.g. +, −, convolution, matmul). 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*.

Most deep learning systems use dataflow graph as the default representation for NNs. In older frameworks such as Caffe [34] and DistBelief [19], the models are composed of existing layers. Modern frameworks such as TensorFlow [1] and Theano [63] represent computation as a dataflow graph of primitive mathematical operations. This representation makes it easy for users to define new layers and new training algorithms using primitive operations. Besides stateless computation operations, the dataflow graph also contains stateful operations such as *variable* and *constant* operations. Besides data inputs and outputs, the computation graph may also contain *control dependency* edges to enforce ordering among operations.

3.2 Distributed Deep Learning and Communication Architectures

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. In a mathematical form, given data $D$ and a loss function $L$, fitting the parameters $\theta$ of a NN can be formulated as an *iterative-convergent* algorithm that repeatedly executing the update equation

$$\theta(t) = \theta(t-1) + \epsilon \cdot \nabla_L(\theta(t-1), D(t))$$

until $\theta$ reaches some stopping criteria, where $t$ denotes the iteration. The update function $\nabla_L$ calculates the gradients of $L$ 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. Leveraging the SIMD nature of modern accelerators (e.g. GPUs and TPUs), a stochastic batch of training samples $D(t) (D(t) \subset D)$ at each iteration $t$
could be *naively parallelized* as fast as processing a single sample, and the accumulation of the gradients over all samples in the batch is used to update the parameters, as in Eq. 1.

In large-scale deep learning, data $D$ are usually too large to process on a single machine in an acceptable period of time. To speed up the training, we partition the data $D$ and distribute to a cluster of computational worker machines (indexed by $p = 1, \cdots, P$), as illustrated in Figure 1. At each iteration $t$, every worker fetches a batch $D_p^{(t)}$ from its data partition and computes the gradients $\nabla L(\theta^{(t)}, D_p^{(t)})$. Gradients from all workers are then aggregated and applied to update $\theta^{(t)}$ to $\theta^{(t+1)}$ following

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

This strategy allows data to be locally partitioned to each worker, which is advantageous for large dataset. It however requires every worker to have read and write access to the shared model parameters $\theta$, which causes communication among workers; this shared access can be provided by a parameter server architecture [11, 67] (Figure 1 left), or via collective communication such as AllReduce [55] or a peer-to-peer broadcasting architecture [69] (Figure 1 right).

**Parameter Server (PS).** A parameter server (PS) is a distributed shared memory system that provides a 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. DL can be trivially parallelized over distributed workers using PS with the following 3 steps: (1) Each worker computes the gradients ($\nabla L$) 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 1a).

**Sufficient Factor Broadcasting (SFB).** 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). Sufficient factor broadcasting (SFB) [69] is designed 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

![Figure 1](image-url)
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.

**Collective AllReduce.** Synchronizing messages across GPU or via RDMA can be significantly accelerated by leveraging mature collective communication libraries such as MPI or NCCL, with their implementations highly optimized against HPC devices. Employing collective allreduce in distributed deep learning is straightforward: an *allreduce* operation is performed to aggregate the gradients across all workers, and then each worker updates their own copy of parameters using the gradient accumulation, maintaining global consistency. Based on how the gradients should be reduced across multiple workers, there are different variants of allreduce algorithms. One of the most adopted implementation is *ring allreduce* [55], which allows worker nodes to average gradients and disperse them to all nodes, and is bandwidth-optimal.

### 3.3 Static and Dynamic Neural Networks

Modern DL has been developed and applied extensively over data with more complicated structures, e.g. data structured as sequences, trees and graphs, which are required to tackle practical problems such as machine translation [60][61], question answering [62], and semantic image segmentation [42][75]. RNNs are a typical class of dynamic 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 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) [33] and gated recurrent units (GRU) [12]. 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 2:** A cell function shown in (a) could be applied on different structures such as a (b) chain (c) tree, or (d) graph.

### 3.4 Consistency Models in Distributed ML

Bulk synchronous parallel (BSP) is a commonly used synchronization mechanism for data-parallel training. Under BSP, workers alternate between computation and synchronization. After computing a local computation step, each worker enters a synchronization phase, and propagates update messages generated from its local computation to other workers and receives others updates. Thus the synchronization phase does not end until all workers finish communicating updates. Such a parallelization model is simple but is prone to poor performance since all workers proceed at the speed of the slowest worker.
In order to overcome the communication overhead and mitigate waiting for stragglers, people also proposed totally asynchronous parallel (TAP). In contrast to BSP, a worker propagates parameter updates and fetches new parameter values (typically from PS) without waiting for other workers. TAP achieves high computation throughput, however, staleness may be arbitrarily large and even lead to divergence.

Motivated by the staleness problems of TAP, stale synchronous parallel (SSP) ensures bounded staleness by blocking a worker’s computation when its locally cached parameter states are more than $T$ steps stale. This also means the fastest worker cannot be more than $T$ steps ahead of the slowest worker. Previous work [16,32] proves that convergence is guaranteed for certain models when step size is properly tuned. In this thesis, we refer staleness as a parameter that controls the degree of consistency in parameter synchronization – when staleness equals 0, we achieve BSP; the consistency model is TAP when staleness is infinitely large.

3.5 Other Terms and Notations

ML Parallelism. A few notable technologies that are studied in this thesis include: node or layer partitioning as for model parallelism [35, 36, 57, 66], various synchronization architecture including parameter server, allreduce, sufficient factor broadcasting, consistency model as introduced in section 3.4, gradient compression techniques, memory management and scheduling for ML, just to name a few. In literature, ML parallelisms generally refer to either data parallelism or model parallelism. In this thesis, we generalize the term ML parallelism to a broader scope. We will call any of these technologies an ML parallelism, even though we note that different parallelization strategies may come from or are designed to address different dimensions of problems for ML parallelization.

Parallelization Strategy. We use strategy (or parallelization strategy, distribution strategy) to refer to an instantiation of one or a combination of multiple parallelisms mentioned above as a mean of ML parallelization. For examples, an AllReduce strategy means choosing collective AllReduce as the synchronization architecture to distribute the ML program; a stale synchronous parameter server strategy means choosing PS as the architecture and stale synchronous as the consistency model to distribute a ML program.

4 Adaptive Parallelism

DL models exhibit a high degree of model complexity, with many parameters in deeply layered structures that usually take days to weeks to train on a GPU-equipped machine. As ML and deep learning models become more structurally complex [22, 31], one monolithic parallelization strategy has struggled to provide excellent all-round performance on a wide variety of models – in practice, each system’s implementation is mostly based on one parallelism, and only well-suited to a limited number of model architectures. For examples, there is evidence to suggest that: bounded-asynchronous parallelism may be better suited to models with shallower depth [18], sufficient factor broadcast is more effective with large matrix-structured parameters and small-to-medium numbers of nodes [70, 80]; bipartite parameter server architectures have been shown to be successful at models whose sparsity structure creates “hot spots” [11, 40]; collective allre-
duce would outperform other communication architectures when the majority of the distributed communications happen between GPUs [26, 55].

The first part of this thesis investigates how each parallelization strategy would perform against different types of ML models or components, considering the model’s ML structure, computational and statistical properties, and the training hardware environment. The thesis reveals that conditioned on these factors, different parallelisms effects very differently. In order to be performant, this thesis proposes a design principle, adaptive parallelism – based on the structures and composability nature of DL models, adaptively and mixedly applies different parallelisms depending on the models, algorithms, and even system configurations.

In our finished and ongoing work, we show that this methodology is instantiated to improve parallel ML in the following aspects: execution scheduling (section 4.1.1), memory management (section 4.1.2), communication (section 4.2), consistency (section 4.3), and resource scheduling (section 4.4).

### 4.1 Adaptive Scheduling and Memory Management

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.

DL software such as TensorFlow [1] and Caffe [34] allows 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 (229M parameters), open-source TensorFlow on 32 machines can be slower than a single machine. This observation underlines the challenges of scaling DL on GPU clusters: (1) 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. The iterative nature of DL algorithms causes the updates to be transmitted in bursts (at the end of an iteration or a batch of data), with significant periods of low network usage in between. Existing communication strategies, such as a vanilla parameter servers (PS) for ML [40, 67], can be overwhelmed by the high volume of communication [14]. (2) The GPU device memory does not grow proportionally with the trend of using larger models to solve more complex problems. The need to fit the full model, as well as a mini-batch of input data and produced intermediate states, in the GPU memory limits the size of models that can be trained.

Fortunately, the layered structure of neural networks offers chances for improvement. We show next that these two challenges can be mitigated by adapting the execution scheduling and memory management to the layered structure of DL models — based on the structured dependencies present in NNs, the computation and communication of different layers of NNs can be rescheduled so as to avoid bursty network traffic, and the GPU memory consumption can be reduced by offloading parameters and activation values to cheaper host memory and loading them only when they are needed. This part of this thesis presents two instantiations of adaptive parallelism with respect to the execution scheduling and memory management in distributed DL: Wait-free back-propagation (WFBP) and memory swapping.
4.1.1 Wait-free Backpropagation (WFBP)

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 Fig. 3. When executing on distributed environment, inter-machine communications are required after each \( C \) step – we similarly define a parameter Synchronization step as \( S_t \). Therefore, a naive parallelization of DL training over distributed GPUs 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 (Fig. 3 upper left). Fortunately, the sequential nature and composability of NNs present us an opportunity to decompose and overlap the computations and communications. 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 or aggregated gradients (depending on the communication architecture chosen) remotely \((i^l_t)\), we can rewrite a training iteration as: \( [C_t, S_t] = [f^1_t, \ldots, f^L_t, b^L_t, \ldots, b^1_t, o^1_t, \ldots, o^L_t, i^1_t, \ldots, i^L_t] \).

The wait-free backpropagation (WFBP) is designed to adapt a vanilla sequential computation-communication schedule to fit with this structure. It is based on two key independence in a DL program: (1) the send-out operation \( o^l_t \) is independent of backward operations \( b^i_t (i < l) \), so they could be executed concurrently without blocking each other; (2) the read-in operation \( i^l_t \) could update the layer parameters as long as \( b^l_t \) was finished, without blocking the subsequent backward operations \( b^i_t (i < l) \). Therefore, we can enforce each layer \( l \) to start its communication once its gradients are generated after \( b^l_t \), so that the time spent on operation \( s^l_t \) could be overlapped with those of \( b^i_t (i < l) \), as shown in the right of Fig. 3.

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.
November 19, 2019
DRAFT

VGG [59] and AdamNet [11, 14]), because it overlaps the communication of top layers (90% of communication time) with the computation of bottom layers (90% of computation time) [14, 78]. 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.

**Results.** We implement the WFBP with a parameter server architecture and show that this rescheduling is key to the performance of distributed DL on GPUs. It can overlap communication with computation effectively, and yield higher utilization of GPU cores, and avoid bursty communication across network switches. With WFBP, improved scalability has been observed in multiple DL frameworks including TensorFlow and Caffe, across multiple NN architectures (i.e. GoogLeNet, VGG, VGG19-22K), and regardless of whether the network bandwidth is abundant or insufficient.

**Related Work.** Most DL frameworks, such as TensorFlow, represent the data dependencies of DL programs within dataflow graphs, and cast the communication-computation scheduling generally as a graph scheduling problem. However, they fail to explore the potential opportunities of parallelization between training 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.

WFBP, as one of the earliest structure-aware scheduling for distributed DL, has inspired a few follow-up work, such as ByteScheduler [52] and PipeDream [30]. 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.

4.1.2 Memory Swapping

The limited size of GPU device memory was viewed as a serious impediment to data parallel DL training, limiting the size of the model to what could fit in a single device memory. We present memory swapping, a simple memory management mechanism that is enabled by the fact that a layer’s computation only depends on the output (and intermediate states) produced by its previous layers, but not that by the entire model.

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 slow down data access. To minimize slowdowns, the proposed memory manager will use separate threads to perform the memcpy operations between CPU and GPUs in the background. In order to pre-fetch the content from CPU to GPU, it will need to know in ad-
vance the sets of parameter data that the application will access. This is enabled by incorporating knowledge of the neural network structure in memory management. For example, in the forward pass of CNNs, once the forward computation \( f_i \) 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 \( f_j (j > i) \), 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 back to GPU, can be performed in advance and overlapped with the backward computation operations \( b_j (j > i) \). In this sense, the memory management is adapted and informed with the dependency structure present in the model, hides data movement overheads, but makes it possible for the training of very large models that otherwise cannot fit in the limited GPU memory.

**Related Work.** The closest work to memory swapping is [8], which proposes to trade computation for memory – it only stores intermediate states at a set of staging nodes (layers) in the NN, and recompute the intermediate states from the staging nodes if they are needed during backprop. This strategy similarly exploits the sequential structure of the NNs as ours, but results in more computation up to an entire forward pass.

### 4.2 Adaptive Communication Architectures: PS vs. AllReduce vs. SFB

For a long time, parameter server has been commonly believed to be the right architecture to synchronize messages for iterative-convergent ML workloads [14, 40, 67], as it fits with the computational pattern of iterative SGD, and allows for flexible consistency models. However, recent advances [26, 55] show that, when parallelizing CNNs under fully synchronous settings, near-linear speedup could be achieved with the *ring-allreduce* algorithm and mature collective communication libraries (e.g. MPI or NCCL) from the high performance computing (HPC) community. On the other hand, when facing large models with dense matrices parameters that exhibit low-rank properties, a peer-to-peer based communication architecture, *sufficient factor broadcasting* (SFB), could significantly reduce the size of messages transferred across network. This part of the thesis studies how each communication architecture would fit with different types of ML models and neural network components, configurations, and hardware environments. Based on that, we present a hybrid communication strategy along with a system implementation, as an instantiation of adaptive parallelism with respect to communication architectures, Hybridize PS and SFB. The idea comes from two observations: first, as revealed in Figure 3, the synchronization operations of each layer \( \{S^l\}_{l=1}^L \) are independent of each other. Their implementations thereby do not have to resort to the same communication architecture, but can be instantiated as any architecture if appropriate; second, the communication overhead can be estimated by measuring the number of parameters needed to be transferred based on the model and cluster configurations, we are able to estimate the communication overhead, and infer the optimal method most suitable with a specific synchronization operation.

Consider training the VGG19 network [59] using a parameter server architecture, the overheads of \( S^l \) could be estimated as follows (Table 1): assume the batch size \( K = 32 \), the number of workers and server nodes \( P_1 = P_2 = 8 \) (assuming parameters are equally partitioned over all PS shards), respectively. On one hand, if \( l \) is an FC layer (with shape \( 4096 \times 4096 \), \( M = N = 4096 \),
Table 1: Estimated communication cost of PS, SFB 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 $K$.

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, so we can directly resort to PS or allreduce. We note 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 $M, N, K, P_1, P_2$ – a performant system needs to take into account these factors and allows to flexibly specialize the appropriate communication method for different parts of a model, and to dynamically adjust the communication method – choose the best available method whenever it results in fewer communication overheads.

We implement the hybrid communication in the Poseidon [78, 80] system. We empirically show that Poseidon constantly delivers linear speedups using up to 32 nodes and limited bandwidth on a variety of neural network, datasets and computation engines, and compares favorably to Adam [11] and Microsoft CNTK.

**Related Work.** A second successful instance of adaptive communication [38] is to mix parameter server with AllReduce [38], 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.

### 4.3 Adaptive Consistency

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 [19, 26, 40]. These works, however, point to seemingly contradictory conclusions on whether non-synchronous execution outperforms synchronous counterparts in terms of absolute convergence, which is mea-
sured by the wall clock time to reach the desired model quality. For deep neural networks, Chilimbi et al. [11], Dean et al. [19] show that fully asynchronous systems achieve high scalability and model quality, but others argue that synchronous training converges faster [14, 80].

The disagreement goes beyond deep learning models: Ho et al. [32], Recht et al. [54], Zhang and Kwok [81], Zinkevich et al. [82] empirically and theoretically show that many algorithms scale effectively under non-synchronous settings, but Hadjis et al. [29], McMahan and Streeter [45], Mitliagkas et al. [47] 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 [6, 11, 13, 17, 32]. 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 [6, 14]. How statistical efficiency and system throughput trade off in distributed systems is far from clear. In particular, we are interested in the following aspects: Do ML algorithms converge under staleness? To what extent does staleness impact the convergence?

In this part of thesis, we study the impact of staleness on distributed ML and show that, different models, or even building blocks of models, have different degrees of robustness against staleness, both in theory and practice. It is necessary to adapt the consistency model, usually implemented as fixed in existing ML systems, based on this property, in order to best trade-off between system throughput and statistical efficiency.

**Results.** We study the impact of staleness on a diverse set of models and deep learning building blocks: convolutional neural networks (CNNs), recurrent neural networks (RNNs), fully-connected 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 [24]. 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.

To gain deeper insights, for deep learning and gradient-based methods we further introduce gradient coherence along the optimization path, and show that gradient coherence is a possible explanation for an algorithm’s sensitivity to staleness. In particular, our theoretical result establishes the $O(1/\sqrt{T})$ convergence rate of the asynchronous SGD in non-convex optimization by exploiting gradient coherence, matching the rate of best-known results [41].

Overall, the effects of staleness are highly problem dependent, influenced by model complexity, choice of the algorithms, the number of workers, the model itself, and different components
of a model, among others. Our experiment and theoretical findings show that many ML methods indeed demonstrate certain robustness against low staleness, and offer opportunities for system optimization. The gleaned insights echo the necessity of introducing adaptiveness in the consistency model — with respect to model specifications, i.e. applying different levels of staleness, to tradeoff between system and convergence performance.

4.4 Ongoing Work: Adaptive Resource Scheduling

As the resource demands of ML increases, efficiently managing available resources becomes a rising challenge. Generic resource managers, such as YARN [65], Rayon [15], and Tetris [27], use request-based policies. A user who submits a job should also provide the amount of resources (ie. GPUs) they want to use for their job, and the scheduler tries to satisfy each resource request for the duration of the job. In order to decide the right amount of a resource, the user must know the marginal performance benefit of allocating each unit of that resource to their job. However, distributed machine learning jobs experience quickly diminishing returns. This point of diminishing returns depends on subtle properties of the model being trained, such as the ratio between the size of gradients generated and the computational complexity of each iteration, and the ability of the ML framework to optimize and overlap communication with computation. Without expert knowledge of the implementation details of the ML framework or prior experimentation, it is extremely challenging for a user to know the resource needs of their job.

The last instantiation of adaptive parallelism, as an ongoing work that will be included in this thesis is in the aspect of resource scheduling – develop batchsize-adaptive resource scheduling for DL workloads on shared clusters.

**Batchsize-adaptive elastic scheduling for DL tasks.** The exact effects of the batch size on statistical convergence is an active area of research. First, larger batch sizes result in lower variance estimates of the true gradient at each step, which can lead to faster convergence to a local minimum. However, this speedup is subject to diminishing returns as the batch size is increased. McCandlish et al. [44] define the gradient noise scale, a point at which increasing the batch size switches from having near-linear convergence speedup to near-zero convergence speedup. Furthermore, this gradient noise scale gradually increases throughout a training job’s lifetime, resulting in a dynamic trade-off between batch size and speed of convergence.

The batch size also affects the generalization ability of the final model. Training using larger batch sizes tends to converge to *sharp minima* with worse generalization to unseen data, while training with smaller batch sizes tend to converge to *flat minima* with better generalization [37]. Nevertheless, techniques for achieving good generalization using large batch sizes have been demonstrated. The rule of linearly scaling the learning rate with the batch size has enabled training certain models with good generalization using large batch sizes [26]. Ben-Nun and Hoefler [4] showed that the common technique of decreasing the learning rate during training is equivalent to increasing the batch size by the same factor, leading to a simple way to improve the scalability of later iterations in SGD.

These insights enable calculating the statistical scalability of SGD. Using this information, a scheduler can make much more informed decisions which adaptively adjusts both resources and batch sizes of each job. One way this can potentially be done is to break down the total utility of allocating a certain amount of to a job into (1) a resource-utility function, which models
the system scalability of the job, and (2) a batch-utility function, which models the statistical scalability of the job. By inferring the resource-utility and batch-utility functions during the run-time of each job, the scheduler may calculate and optimize for the joint total utility across all jobs in the cluster. We are investigating this breakdown of the total utility to design an adaptive scheduler for DL which accounts for statistical scalability in its scheduling decisions.

Related work. A few schedulers dedicated for ML training have been proposed. They typically optimize system efficiency while keeping the statistical efficiency fixed. Optimus [51] fits a performance model to each job during run-time, and uses JCT predictions to more efficiently re-allocate resources between concurrent jobs. Gandiva [68] uses run-time traces to time-slice GPUs and dynamically migrate replicas. Tiresias [28], on the other hand, achieves state-of-the-art JCT and makespan assuming static (instead of allowing dynamic) resources and without relying on JCT predictions. Such schedulers treat training parameters (i.e. batch size, learning rate, momentum) as black boxes and do not modify them throughout the lifetime of each job. This allows the scheduler to focus on optimizing system efficiency without impacting the statistical efficiency of each training job. However, we argue that a cluster resource scheduler can achieve more efficient resource allocation for shared-cluster distributed training jobs by coupling its scheduling decisions with each job’s training parameters. In particular, the batch size parameter heavily impacts both the system efficiency and statistical efficiency of each job. The recent research trend in large-batch training [26, 37, 44] reveals that, under many circumstances, the batch size can be increased so that both system scalability and statistical efficiency (per-iteration) are greatly improved. Thus, only optimizing system efficiency results in missed opportunities to achieve the best total efficiency in terms of time-to-convergence. We will explore and design batchsize-adaptive resource management systems that adapts to these training parameters, and leverage these specific properties of ML for better resource management.

4.5 Application: Scaling Up Large-scale Image Classification

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) [21, 56, 74]. This architecture, which is also called HD-CNN [74] (as shown in Figure 4), is however hindered by two system challenges. Introducing multi-level category hierarchies significantly deepens the backbone CNN with more feature learning layers and intermediate classifiers. Each added layer would generate more intermediate states that need to be stored for backprop, increasing peak memory 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, burdening the network communication in distributed training.

We show that scaling up HD-CNN is a perfect examination of the effectiveness of the proposed adaptive parallelism. leveraging the proposed adaptive parallelization strategies, to reduce
Figure 4: (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.

the peak GPU memory usage, we let the system swap inactive intermediate states back to host memory based on the model structure, and reserve space on GPU memory for the upcoming computation. To communicate the parameters of HD-CNN, we customize the communication scheme using hybrid communication based on the layer type and classifier size, effectively minimizing size of messages to be transferred from more softmax layers.

Results. We incorporate a two-layered coarse-to-fine category hierarchy on ImageNet-1000 [20], using VGG-19 [58] as the build block model, resulting in 4x more parameters. We successfully scaled out this model onto 32 GPU-nodes, each with (12Gb) standard device memory. We reported state-of-the-art image classification performance on ImageNet back in 2015 [74]. The echoes that designing adaptive parallelization strategies and factoring the parallelisms with respect to the model specifications not only improve the application performance, but also expand the design space where larger and more complex model architectures could be explored.

5 Representations of Adaptive Parallelisms

With the multiple systems and strategy instantiations presented in the first part of this thesis (section 4), we conclude a few design principles implied by adaptive parallelisms:

- **Composable**: A successful distribution strategy for a complex model usually requires customizing a suitable distribution strategy for each model building block. In this sense, the parallelization strategy itself exhibits composability as the model has, in the sense that the overall strategy appears as a combination of distribution strategies for all building blocks of the model.

- **Model-centric**: Since the strategy is strongly conditioned on the model, the system design with adaptive parallelisms is preferred to be model-centric, instead of being tailored for a single parallelism or strategy (which is the cause of why many existing distributed ML systems are less flexible and only suitable for a narrow family of models). Therefore, we hope the system to be compatible with as many basic parallelisms as possible, so as to adapt to various incoming models and provide excellent all-round performance.

- **Multi-dimensional**: Performant strategies need to be optimized with respect to different
aspects of the parallelization process (or, different dimensions of parallelisms) – ranging from parameter synchronization, consistency model, to execution scheduling, and device placement [46], which we notate as the multi-dimensionality of adaptive strategies.

- **Resource-aware**: the hardware and resource configurations of the cluster where the workload will be deployed impact the choice and optimality of the distribution strategy.

In the second part of this thesis, we ask, starting from these design principles, whether it is possible to generalize those instantiations of adaptive parallelisms, to have systems that implement various parallelisms from different aspects, and for a given model, automatically derive the appropriate distribution strategy fitting with the statistical properties of the model and the hardware configurations of the distributed computing environment. We believe the first step to realize such a system is to obtain good representations of distribution strategies – that, conditioned on the building block representation of deep learning models, are capable of expressing a profound set of ML parallelisms originated from different aspects (synchronization architecture, node/layer partitioning and placement, consistency control, gradient compression, etc.) in a unified way.

To this end, this second part of the thesis will separately discuss two classes of parallel training workloads: distributed parallel training for static neural networks (section 5.1) and single-node batched training for dynamic neural networks (section 5.2), with the topics spanning from the design of the strategy representation, to the derived programming interface, and detailed system abstraction and implementations.

### 5.1 Representation of Distribution Strategies in Static Neural Networks

#### 5.1.1 Representation

We assume the model to be distributed could be represented as a static dataflow graph $\mathcal{G} = \{V_G, E_G\}$, where each node $v \in V_G$ is a computational operation (e.g. `conv2d`, `matmul`) or a variable [1], and each edge $e \in E_G$ represents a multi-dimensional tensor (e.g. intermediate states). Our goal is to parallelize the execution of $\mathcal{G}$ over a computational cluster, represented as another graph $\mathcal{D} = \{V_D, E_D\}$, where each node $d \in V_D$ is a computing device (GPUs or CPUs), and each edge $(d_i, d_j) \in E_D$ is the available communication bandwidth between device $d_i$ and $d_j$.

We are interested in generating a distribution strategy conditioned on the explicit representations of the model $\mathcal{G}$ and the resource specification $\mathcal{D}$, that maximizes the training throughput of $\mathcal{G}$ over $\mathcal{D}$. We start from considering the following set of parallelism dimensions (but will expand the set gradually to include more base parallelisms): (1) node partitioning, (2) placement of computational nodes, (3) dataflow graph replication, (4) parameter synchronization. We define a *distribution strategy* as a description that expresses how this graph should be parallelized with respect to each of these parallelism dimensions, elaborated below.

**Node Partitioning.** Node partitioning usually subsumes most model parallelisms in previous distributed ML literature, and beyond [36]. For computational operations, similar to Tofu [66], we only consider partitioning patterns that follow “*partition-n-reduce*”. Suppose there are $k_i$ possible axes along which the computation described by node $v_i$ could be partitioned and dis-
we then represent a specific partition strategy for \( v \) as a \( k_i \)-dim vector \( p_i = [p_{ij}^{k_i}]_{j=1}^{k_i} \), where each \( p_{ij}^{k_i} \) is an integer as the degree indicating how many partitions will be generated under the \( j \)th axis. Different from a computational operation, since a variable node does not describe specific computational kernels, it could be partitioned flexibly along any axis of its memory buffer. We represent the partitioning of a variable operation \( v_i \) as a vector \( p_i = [p_{ij}^{k_i}]_{j=1}^{k_i} \) where \( k_i \) is the number of dimensions of the tensor buffer of \( v_i \), and each value \( p_{ij}^{k_i} \) is an integer representing how many partitions are generated along the \( j \)th axis.

In this way, we can describe how each node \( v_i \in V_G \) should be partitioned using a \( k_i \)-dimensional vector \( p_i \), where each dimension of \( p_i \) describes how to partition \( v_i \) (computation or variable) along the a specific axis (or, partitioning dimension). We call an instantiation of \( p_i \) as a partitioning strategy of \( v_i \). Combining \( p_i \) for all nodes of \( G \) provides us a description how the entire model \( G \) should be partitioned, which we call the partitioning strategy of the model.

**Node placement.** After \( G \) is partitioned, we figure out the placement of each node in the partitioned graph \( G' \) on the device graph \( D \) – an explored problem called device placement [46]. We can obtain the placement for each node of the graph as a placement representation.

**Replication.** We additionally introduce a parallelism dimension called replication to express how \( G \) should be replicated across multiple devices. This usually maps to the pure “data parallelisms” in many existing distributed ML systems – where all computational operations are replicated but the training data are partitioned. We use a pair \( (G, \{d_i\}) \) to notate that \( G \) will be replicated on a set of destination devices \( \{d_i\} \subset V_D \). Hence we obtain a representation about how to replicate, and any instantiation in the space of the representation is a replication strategy. Note that any replication strategy of \( G \) over \( D \) could be equivalently and implicitly described by a partitioning strategy, that partitions all computational nodes of \( G \) evenly only along the batch dimension and dispatches them onto replica devices. However, we still find that introducing a dedicate description for replication greatly simplifies both the strategy representation itself and some low-level system implementation details, especially in cases where only data parallelisms are interested.

**Synchronization.** Distributed training requires parameter synchronization between devices or cluster nodes. We introduce a parallelism that describes ways of synchronizing parameters when required [38, 55, 80], which we believe shall adapt to the model and resource specifications as argued in section 4.2. For parameter variables that involve applying gradients to update the

---

**Figure 5:** An example synchronization strategy of a variable. It describes that this variable should be synchronized via the parameter server scheme, the variable will be sharded and distributed to a set of devices in “min_ps”. The consistency model shall be BSP (staleness is 0) and each worker node should keep a local replica of the variable buffer to reduce communication.

1Legitimate ways of partitioning a specific computational kernel could either be manually defined [36, 57] or automatically discovered [66], which is an active field of research but out of the scope of this thesis.
parameters, the synchronization configuration considers a few participating factors: (1) the communication architecture: parameter server, allreduce or SFB, (2) gradient compression: whether and how to compress messages when communicating, (3) staleness that controls the consistency model for synchronization, (4) parameter sharding as in PS or (5) reduction structures as in allreduce. They together form a synchronization strategy for a variable. An example synchronization strategy is illustrated in Figure 5.

So far we have covered the strategy representation for each parallelism dimension we considered. Naturally, combining the strategy representations for all parallelism dimensions gives a full description of how this model shall be parallelized over distributed devices – we obtain the distribution strategy of this model. We argue that deriving such as unified representation enclosing aforementioned parallelisms offers a lot of advantages:

- **Principled representation for adaptive parallelisms.** This representation allows to map a model component to its suitable parallelism, and represents the most appropriate parallelization strategy of the model entity as compositions from multiple single-dimensional parallelisms, providing a principled representation for adaptive parallelisms.

- **Simplify distributed parallel ML programming.** It helps simplify distributed parallel ML programming, as the semantics about how an ML model shall be parallelized is totally captured by this representation, isolated from users’ ML prototyping code.

- **Decoupled and extensible system implementations.** A composable representation of distribution strategies guides the design and abstraction of system implementation to be less strategy-centric. This allows low-level detailed system implementations to decouple from one or two specific parallelism, avoids re-implementing specialized systems when new models or parallelism is introduced, unifies multiple existing parallel ML systems, and improves extensibility.

- **Define Search Space for Automatic Parallelization.** Such a representation spans a strategy space including all possible distribution strategies conditioned on the given model. It enables automatic search over the space for the optimal strategy, which will be discussed in the third part of this thesis.

### 5.1.2 Existing Work

Some existing works [36, 46, 57, 66] have investigated the problem of graph partitioning or device placement. Among them, closest to ours are FlexFlow [36] and Tofu [66]. Both works focus on graph partitioning, and propose algorithms to partition each individual layer or operation of the model. For a given layer of the neural network, FlexFlow [35, 36] defines four partitioning dimensions “SOAP”, and uses a stochastic MCMC-based algorithm to search for the optimal partitioning strategy over the entire model. FlexFlow partitions at the granularity of DNN layers, with a simpler and less expressive searching space than operations in dataflow computational graphs. Tofu [66] improves over FlexFlow. Instead of defining the legitimate partitioning of an operation within the SOAP space, they automatically discover the possible partitioning dimensions of a computational kernel using symbolic interval analysis; then, they propose a few optimizations that reduce the search space, and use the same DP algorithm as in [35] to search
for the best partitioning strategy of the entire graph. We go one step further, and focus on deriving a more generic representation to cover more aspects of parallelisms, such as replication, synchronization, consistency model and message compression.

5.1.3 System Implementation

We develop a system, Arion, based on this proposed representation. Arion’s system implementation is based on dataflow graph rewriting. Given a dataflow graph declared by users that run on a single machine (i.e. without semantics about distributed execution), Arion incorporates distribution semantics and coordination logic by adding, deleting or modifying nodes and edges in the dataflow graph. Each base parallelism therefore maps to a graph writing implementation, called graph transformation kernel, that rewrites the graph from its current state to the next state so that the semantics of the parallelism is present in the new graph. Arion implements a library of graph writing kernels – partitioning kernels, that partition and dispatch operations or variables following a partitioning strategy \cite{36, 57}; replication kernels that replicate the graph across devices; placement kernels that assign operation to designated hosting device; synchronization kernels that, depending on the specified parameter synchronization scheme, alter the graph to share variables and pass gradients (PS), or perform collective communication (all-reduce) or gradient compression etc.

Hence, distributing a single-node dataflow graph using Arion is essentially applying a series of graph transformation kernels provided by Arion following the distribution strategy expression. This allows the lower-level system implementation to focus on providing various transformation kernels that are composable and cover a diverse set of parallelisms. Arion delegate the execution of the dataflow graph to framework runtime – so that the distribution logic stays agnostic to ML frameworks, and respects their existing programming interface.

For ML practitioners, using an exiting strategy to distribute their ML programs is done by simply offering Arion the computational graph representation of the model and a desired StrategyBuilder (e.g. parameter server). Following the StrategyBuilder, Arion will derive a strategy expression conditioned on the computational graph. The expression encloses details about how each operation or variable should be partitioned or synchronized using proposed strategy representations, as explained in section 5.1. Then the system applies the strategy expression on original dataflow graph – rewriting the graph using the corresponded graph transformation kernels. Strategy developers can invent new or better parallelization strategies by implementing new StrategyBuilders, instructing how to generate a strategy expression conditioned on the computational graph – which is usually done by assembling base parallelisms covered in our representation. Support a base parallelism that is missing in the representation can be done by adding its corresponded graph transformation kernels into Arion backend.

5.2 Representation of Parallelisms in Dynamic Neural Networks

We next shift our focus to the representation of DL models with static structures and their parallel execution. While a key assumption about whether the computational graph of the model would change with incoming training distinguishes this class of workload and the design of representations from those discussed in section 5.1, we note the parallelization techniques developed for
Figure 6: The workflows of (a) static declaration, (b) dynamic declaration, (c) Cavs’ vertex-centric programming model. Without ambiguity, we use $D$ to denote both the dataflow graph itself and the computational function implied by $D$.

these two workloads are not totally exclusive, as there exist a great amount of dynamic neural networks in which their input-dependent structures could be captured and represented using symbolic control flow operations [76], which is static and can benefit from any technical development assuming static model structures.

5.2.1 Background: Static Declaration and Batching

One dominant paradigm in the training of DL models, adopted by toolkits such as Caffe and TensorFlow, uses static dataflow graphs [1, 48]. These graphs represent the flow of data through computational functions, and are defined using symbolic programming [1, 5], 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 [3]. Figure 6(a) summarizes the programming model derived from these dataflow graphs, which is named as static declaration and has been adopted in many DL frameworks [1, 5, 7].

Parallelization. The computation of multiple data samples in a static dataflow graph can be naturally parallelized (or batched) – at each iteration $p$, a batched tensor of $K$ samples $\{x^p_k\}_{k=1}^K$ is fed to $D$, and the computation is executed in a single pass, allowing for efficient use of memory caches and the SIMD nature of modern hardware (e.g. GPUs), which is extremely advantageous for DL workloads [39]. On the other hand, this parallelization assumes the dataflow graph $D$ is static for all samples and fixed throughout the computation. While it is effective on a wide range of NN models, such as CNNs over fixed-size images, it is not able to describe models or their parallelization with dynamically changing structures, unless resorting to symbolic control flow operations [76].

5.2.2 Dynamic Declaration and Dynamic Batching

With the increasing complexity of the problems to be addressed, DL has been extended and applied on data with more complicated structures, such as sequences [33, 60], trees [61] and graphs [42], over which the NN may conditionally choose its own computation order for specific modeling needs, i.e. the structure of its dataflow graph changes over training.

For this family of neural networks, a programming model explored in this thesis is called dynamic declaration [49, 64]. Figure 6(b) contrasts dynamic declaration with static declaration. It allows the user to: (1) define a different computation graph for each training example, 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.

**Parallelization.** Dynamic declaration, as a feasible representation of the computation for dynamic NNs, is not friendly to parallelization, as it creates an instance-dependent dataflow graph per training sample, resulting in many differently shaped dataflow graphs, which prevents the natural batched execution commonly seen in static declaration. Dynamic batching [50] is therefore proposed to seek chances of parallelization. It analyzes all the (differently shaped) computational graphs defined for each sample within a batch, and identifies batch-able nodes in the graphs, such as operations with exactly the same signature, input and output shapes, and groups them for batched execution.

**System Implementations.** We contributed to DyNet [25], an open sourced system implementation for dynamic declaration. DyNet is designed to minimize the cost of graph construction, as usually criticized as a major disadvantage of dynamic declaration. While dynamic declaration has sufficient expressiveness to describe arbitrarily dynamic NNs, it faces a few challenges: (1) since the dataflow graphs are always changing, execution based on the representation of multiple dataflow graphs can hardly benefit from any well-established dataflow graph optimization techniques [1, 9, 53, 79] – graph post-processing and optimizations have to be performed for each dataflow graph, but only benefiting for a single sample and backprop pass. (2) At the backend, since a dataflow graph needs to be constructed per sample, the overhead is linearly increasing with the number of samples, and is not negotiable even in systems with optimized graph construction [25], and sometimes yields downgraded performance [43].

### 5.2.3 Vertex-centric Representation and Cavs

**Representation.** Alternatively, we present a vertex-centric representation [79] that facilitates the programming of dynamic neural networks and their efficient batching. The motivation comes from several key principles ML developers usually comply with to ensure the feasibility and learnability of the model during their design of dynamic NNs. We note most dynamic NNs are designed to exhibit a recursive structure (e.g. sequence RNN, Tree-RNN), or a combination of static and recursive structures (e.g. LRCN [2, 23], attention [73]), or even a combination of multiple recursive structures (e.g. encoder-decoder RNNs [60]).

To explain the vertex-centric representation, we call the instance-specific structure associated with the input sample as an **input graph**, and notate it as \( G \), and a node in that graph as a **vertex**, to be distinguished from a dataflow graph \( D \) and the nodes (which are usually operators or variables) therein. The vertex-centric representation is inspired by a core observation: Within one specific recursive structure, a static function is dynamically applied over instance-specific graphs; in the instance-specific graph, every vertex of it computes and interacts in the same way with other vertices, as defined in the function. The function, however, is defined once before the execution, and fixed throughout iterations and across data samples. Figure [7] illustrates the concept of this vertex-centric programming model. To describe a dynamic neural network, different from dynamic declaration, which requires users to manually declare dataflow graphs for each sample according to its associated graph, the representation does not require defining the overall
Figure 7: We represent a dynamic neural network as a static vertex function $\mathcal{F}$ (right) being applied over a dynamic input graph $\mathcal{G}$ (left) associated with the data sample.

Figure 8: An illustration of batched execution with vertex-centric representation. The evaluation of the vertex function $\mathcal{F}$ is batched on the group of nodes with the same color.

dataflow graph explicitly. To be aware of what computation shall be performed, we instead ask users to implement a simple vertex function $\mathcal{F}$ by “thinking like a vertex”, informing the system how the static function shall compute statically, and exchange the results with its connected vertices (if these is any) in $\mathcal{G}$. In $\mathcal{F}$, users can utilize conventional DL operators to assemble a symbolic construct that will be evaluated dynamically following the structure of $\mathcal{G}$, while the low-level system will ensure the correctness and efficiency. Therefore, a vertex function $\mathcal{F}$, when unrolled across the input structure $\mathcal{G}$, is equivalent with the full dataflow graph of the model that may need to be explicitly declared in static or dynamic declaration.

Accordingly, we present a corresponded programming model, which we call vertex-centric dataflow graph programming, derived based on this representation in Figure 6c, as a comparison with static and dynamic declaration.

Parallelization. The vertex-centric model transforms the batching of multiple dataflow graphs with different structures [43, 50] into a parallelization-friendly representation – scheduling the execution of the vertex function following the input graph structures, as illustrated in Figure 8. For the latter representation, because $\mathcal{F}$ has fixed shape, we can easily batch the execution of $\mathcal{F}$ over multiple vertices simply following the graph dependencies, leveraging the batching capability of GPUs. Moreover, as the vertex function itself maps to a static symbolic dataflow graph, it is open to various graph optimization techniques originally developed for static declaration, such as kernel fusion, streaming, and a proposed lazy batching technique [79]; once performed,
the optimization will benefit all samples and computational iterations – which is not the case in dynamic declaration.

**System Implementation.** We present Cavs, a system implementation based on the proposed vertex-centric representation. Besides the generic math operators in common ML frameworks used to declare the computation, Cavs exposes four symbolic APIs for users to specify how the messages shall be passed between vertices in the vertex function: `gather`, `scatter`, `pull`, `push`. These simplify user programs and bypass the overhead of repeated dataflow graph construction. Cavs implements a scheduler to parallelize the execution of $F$ on multiple vertices during the evaluation of a batch of samples with different input graphs, and a novel memory management mechanism to guarantee the memory coalescing. Together they yield significant performance improvement – we compare Cavs to TensorFlow Fold [43] and DyNet [49, 50], and verify that both Fold and DyNet suffer from substantial overhead caused by repeated graph preprocessing or construction, which is bypassed by Cavs. In terms of the overall performance, on static NNs, Cavs demonstrates equivalent or slightly better performance than Fold and DyNet, while on several dynamic NNs with notably difficult-to-batch workloads (e.g. Tree-LSTM [61] and Tree-FC [43]), Cavs demonstrates near one order of magnitude speedups across various datasets and hyperparameter settings.

## 6 Automatic Search for Adaptive Parallelisms

This third part of the thesis is planed to explore topics on using learning-based methods for automatic distribution strategy search. This echoes the recent trend of *machine learning for system* that uses ML techniques to aid or automate system design.

### 6.1 Proposed Work: Distributed Strategy Search

In almost all existing systems for parallel ML, the burden of deciding the right distribution strategy is placed on the ML researcher or practitioner, which in turn requires ML users to understand the trade-offs inherent in these systems, which is unrealistic.

There has been growing interest in employing learning-based methods to improve system design [10, 36, 46, 72]. Mirhoseini et al. [46] use reinforcement learning to predict device placement for nodes in a TensorFlow computational graph. Jia et al. [36] propose a SOAP representation of parallelisms, and use a stochastic MCMC-based algorithm to search for the optimal partitioning strategy given the graph. Chen et al. [10] built an ML based cost model to guide the code generation for device-specific computational kernels in machine learning frameworks.

Based on the proposed strategy representation, there are chances of enhancing Arion with learning-based approaches to auto-generate distribution strategies for unseen models. Some major research questions we want to explore are as follows.

**Prune the search space.** The parallelism representation defined in section 5.1 spans a large search space. Searching by enumerating every possible instantiation therein would be impossible. Reinforcement learning based methods, as in [46, 83], demands days or weeks of training time on hundreds or thousands of computational resources. We want to incorporate existing human knowledge (e.g. how a certain type of layer/model shall be distributed, in what situation a
parallelism is advantageous) to prune the search space and accelerate the search.

**Reducing search/learning cost.** Once a strategy expression is generated, applying this strategy to the model and performing a trial run on a large distributed cluster would take seconds or minutes. Depending on how many data are needed to train the meta model, performing trial runs in order to collect sufficient training data for meta learning could be resource-demanding. We want to design cost models and simulators so that the performance of similar strategy representations could be directly estimated, reducing searching or learning costs.

**Generalization.** We want to investigate the generalization ability of learned meta models and make sure the trained model can generalize to unseen models, unseen resource specs, with a low inference cost.

### 6.2 Proposed Application: Auto Parallelization of BERT

The testbed application we are interested in exploration is to automatically scale up the training of deep bidirectional transformers (BERT) [22]. BERT and its follow-up variants have recently gained increasing attentions from the natural language processing community by surpassing human performance on a lot language understanding tasks, and become a backbone for many NLP applications. However, unlike successful vision models (e.g. ResNet [31]), training BERT is less accessible to most researchers in academia or practitioners in industry, as it can take weeks on thousands of GPUs or TPs, which hinders continued research and innovations. We have investigated most existing open source implementations of BERT that allow for training from scratch, and noted that most training strategies are based on a standalone parameter server or AllReduce architecture – an improved distribution strategy could be composed or searched based on our proposed representation and there might be space that the training could be made more accessible with commodity resources. Below are a few optimization opportunities we want to investigate:

**Exploring flexible staleness.** BERT model generally trains with a gigantic vocabulary that maps to a huge embedding matrix, occupying the majority of the model parameters. Our study on the impacts of staleness (section 3.4) reveals that sparse variable like embedding matrices has strong robustness to staleness in distributed training. The proposed representation is capable of expressing strategies that has varying staleness for different components of the model – we want to explore such strategies in the distributed training of BERT.

**Overcoming Diminishing returns.** Our initial experiments reveal that when scaling BERT using data parallelism up to a certain amount of replicas, the scalability gains on convergence diminishes quickly. We want to investigate the causes behind both systematically and algorithmically.

**Model compression for distributed training.** There are successful examples in compressing pre-train BERT models to a smaller one using distillation or other techniques without performance drop. This implies that BERT has an over-expressive parameter space, offering chances for introducing parameter compression to reduce synchronization cost and improve distributed training performance.

### 7 Thesis Timeline

- **Jan 2020**: Release the Arion system.
- **Now - April 2020**: Continue developing Arion backend, and experiment on automatic distribution strategy search using BERT as the application.

- **May 2020**: Submit the Arion system paper and automatic strategy search paper.

- **May - June 2020**: Thesis writing and defense.

References


computer vision and pattern recognition, pages 2625–2634, 2015.


[60] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Advances in neural information processing systems, pages 3104–3112, 2014. 3.3 5.2.2 5.2.3


[67] 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. 3.2, 4.1, 4.2


[69] Pengtao Xie, Jin Kyu Kim, Yi Zhou, Qirong Ho, Abhimanu Kumar, Yaoliang Yu, and Eric Xing. Distributed Machine Learning via Sufficient Factor Broadcasting. In arXiv, 2015. 3.2

[70] Pengtao Xie, Jin Kyu Kim, Qirong Ho, Yaoliang Yu, and Eric Xing. Orpheus: Efficient distributed machine learning via system and algorithm co-design. 2018. 4


[76] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, et al. Dynamic control flow in large-


