Identifying the Limits of Parallelism in Large-scale DNN Training: A Model-Driven Analysis

Albert Kahira*‡,Truong Thao Nguyen†‡,Mohamed Wahib†,Leonardo Bautista Gomez*,Rosa M Badia*,Ryousei Takano†
*Barcelona Supercomputing Center, Barcelona, Spain
† National Institute of Advanced Industrial Science and Technology, Japan

***Abstract*—Deep Neural Network (DNN) frameworks use distributed training to enable faster time to convergence and alleviate memory capacity limitations that come along when training large models, high dimension inputs, and large mini-batch sizes. With the steady increase in datasets and model sizes, model/hybrid parallelism is deemed to have an important role in the future of distributed training of DNNs. We analyze the compute, communication, and memory requirements to understand the tradeoffs of parallelism choices on performance. Our work aims to be the basis for an analytical performance model that aids users, framework programmers, and system designers in identifying the challenges and limitations when choosing a parallel strategy, an implementation, or an architecture design. We demonstrate the utility of the model in detecting limiters and bottlenecks by using experiments with different datasets and models with up to 1000s of GPUs.**

I. Introduction

In recent years, Deep Neural Networks (DNNs) have achieved outstanding results in a wide range of applications, including language processing, speech, and visual recognition. In the quest to increase solution accuracy, there are trends of increasing the size of training datasets as well as introducing larger and deeper DNN models [1], [2], [3]. In addition, applying Deep Learning (DL) in new domains, such as health care and scientific simulations, introduce bigger data samples and more complex DNN models [4]. Those trends make the DNN training computationally expensive for a single node. Therefore, large-scale parallel training on High Performance Computing (HPC) systems or clusters of GPUs is becoming increasingly common to achieve faster training time for larger models and datasets [1]. When training a specific DNN model on an HPC system, there are two prominent strategies for parallelizing the training phase of DL: data and model parallelism. It is important to note that despite the early investigation of model parallelism in DL [5], those efforts were premature and remained far from production deployments, since data parallelism was simple and sufficient.

Scaling data parallelism can be limited by the memory capacity, communication overhead, and mini-batch size. *The first issue* with data parallelism is that the entire model is duplicated for each compute node. Therefore, training larger and deeper neural networks have to deal with the memory capacity limits. For example, [3] reported that GPUs memory has only increased from 12 GB in 2014 to 40 GB in 2020 while the DL models increased approximately 36x in terms of the number of parameters. A notable case is in the area of language modeling and NLP at which models are increasingly approaching $O(100B)$ parameters [6] (ex: GPT-3 has 175B parameters [7]). In addition, as the sample size becomes larger, e.g., more dimensions, higher resolution images, the memory capacity also limit the number of samples that can be concurrently processed by a GPU [8]. Hence, restricting the scaling of data parallelism. *The second issue* in scaling data parallelism is the large communication overhead for gradient exchange among computing nodes at the end of each iteration. As the number of computing nodes increases, e.g., up to 2048 GPUs for Resnet50 as in [9], [10], the communication becomes a bottleneck due to the Allreduce collective operation. Several active efforts try to optimize the reduction collective algorithm for supporting large messages, when considering the specific network architectures in HPC systems [10], [11], [12]. However, even with those algorithms, communication still becomes a bottleneck when the size of DL models increase (e.g., up to $600M$ and $1.5B$ parameters in AmoebaNetB(6,512) [3] and GPT-2 [13] respectively). *The third issue* is the degradation in convergence accuracy when the mini-batch size increases [14], [15]. In recent years, many successful efforts have been dedicated to increase the mini-batch size without (or with small) loss in accuracy [16], [17], [9], [10]. For example, work in [10] reports that the mini-batch size reach up to 81K, when training ImageNet [18] on Resnet50 using Stochastic Gradient Descent (SGD) with up to 2048 GPUs. It helps to reduce the training time from 29 hours, on a single V100 GPU, down to 74.7 seconds [10]. However, there is no empirical evidence or theoretical proof that ensures that techniques such as LARS [19] and LAMB [20], can work well with other different datasets and DL models.

Given these issues on data parallelism and the constantly growing scale of DL training, researchers have started to tackle different bottlenecks across the different components necessary for distributed DNN training. We do an in-depth analysis of those bottlenecks, and discuss how different proposals attempt to circumvent those bottlenecks. Table I shows a summary of the recent approaches in scaling distributed DNN training, divided down by components and training phases. Despite these efforts, data parallelism is not enough for all cases. Thus, it is important to study how model parallelism can achieve good scaling for those large models/datasets. In this work, we focus on the HPC aspects of scaling model parallelism. Innovations in DL theory/practice itself are out

§‡Both authors contributed equally to this research.

TABLE I: Recent progress (and examples) in scaling distributed training of DNNs, across different training components and phases. Training components: **AP**- application (models and datasets), **TA**-training algorithms, **PS**-parallel strategies (computation and communication), **FR**-framework and **SY**-computer systems. Training phases: **IO**-I/O and pre-processing, **FB**-a forward and backward propagation, **GE**-the gradient exchange (if needed) and **WU**-updating the weights. ●: related components. ✓: related training phases. Explanation in remarks.

Approaches	Components					Training Phases				Additional Remarks
	AP	TA	PS	FR	SY	IO	FB	GE	WU	
Optimization methods	○	●	○	○	○	-	-	-	✓	Second order methods [21] (fewer epochs to converge, but longer iterations and more memory).
Normalization	●	○	○	○	○	-	✓	-	-	Cross-GPU Batch-normalization [22] (requires extra communication) and Group Normalization [23].
Pre-trained model	●	●	○	○	○	-	✓	-	-	A big model, that is pre-trained on a big generic dataset, such as Google BiT [24] can be fine-tuned for any task, even if only few labeled samples are available.
Allreduce optimization	○	○	●	○	●	-	-	✓	-	Reduce the communication time by considering the specific network architectures of HPC systems [10], [11], [12] (data parallelism)
Sparsification	○	●	●	○	○	-	✓	✓	✓	Reducing the computation volume [25] or/and communication message size [26] by skipping the computing/transferring of non-important weights/gradients, instead only performing on the significant ones (mainly for data parallelism).
Memory optimization	○	○	○	●	○	-	✓	✓	✓	Reduce required memory by using lower precision (quantization) [27], gradient checking point [28], out-of-core methods [29],
Network architecture	○	○	○	○	●	✓	-	✓	-	Increasing number of GPUs intra node, i.e., up to 16 GPUs in DGX-2. High-throughput inter-node network topology such as HyperX [30] or BiGraph [31].
Model/hybrid parallelism (our target in this work)	○	○	●	○	○	✓	✓	✓	✓	Castello et al. [32] analyzed the communication trade-offs in some model parallel strategies. Hybrid parallelism are proposed in [33] (spatial with data) and [34] (channel/filter with data). [35], [36] explored different parallelization schemes on per-layer basis.

of the scope of this paper. More specifically, we study the practical limits and bottlenecks of different parallel strategies. In addition, we propose a model-driven approach to quantify and detect the effectiveness of optimization approaches at different training phases, and trade-offs that emerge.

Our main contributions in this work are as follow:

- We formally define the main parallel strategies (See Section III) and provide a comprehensive analysis of the performance and memory footprint when training DNNs. To our knowledge, this is the first work to base the analysis on optimizing for memory footprint and not just performance.
- We propose a model that projects the ideal performance of training DL models, broken down by training phases. The ideal performance can aid in favoring a parallel strategy on a given system (See Section IV) and helps to identify optimization opportunities in frameworks.
- We use our model and empirical evaluation to identify the bottlenecks and limitations (Section V).

II. Background

A. Components and phases of Distributed Training of DNNs

1) Stages of Distributed Training: DNNs are made up of a network of neurons (represented as nodes) that are organized in layers (a model). A DNN is trained by iteratively updating the weights of connections between layers in order to reduce the error in prediction of labelled datasets. That is, for a given dataset of D samples, a DNN is trained to find out the model weights w for which the loss function L is minimized. Distributed training of a DNN can be divided into four phases: (**IO**) I/O and pre-processing, (**FB**) a forward phase at which the samples pass through the entire network, followed by a backward phase (back propagation) to compute the gradients, (**GE**) the gradient exchange (if needed) and (**WU**) updating the weights. Specifically, the samples are picked up from the dataset randomly in batches of size B (mini-batch). The training process is then performed on those batches of samples iteratively by using an optimization algorithm such as the Stochastic Gradient Descent (SGD), in which, weights are updated with a learning rate ρ via $w^{iter+1} \leftarrow w^{iter} - \rho \frac{1}{B} \sum_{i \in Batch} \left(\frac{dL}{dw}\right)_i$. The process is then repeated, until convergence, in epochs that randomize the order at which the input is fed to the network.

2) Components of Distributed Training: To optimize for the performance and efficiency of training in large-scale, researchers introduce improvements to the methods, algorithms and design across entire training components which includes: **AP**- application (Deep Learning models and datasets), **TA**-training algorithms (ex: SGD or second order methods), **PA**-parallel strategies (model of computation and communication), **FR**-framework and **SY**-computer systems.

B. Notation

We summarized our notation in Table II. In a G-layer DNN model, a convolution layer l mainly needs these tensors:

- The input of layer l with N samples, each sample include C_l channels, each channel is a tuple of d-dimension: $x_l[N, C_l, X_l^d]$. In a 2-dimension layer, we replace X_l^d with $[W_l, H_l]$, i.e., $x[N, C_l, W_l \times H_l]$. In a clear context, we omit the layer index l and the dimension d, i.e., $x[N, C, X]$.
- The output (activation) of layer l with N samples and F_l output channel $y_l[N, F_l, Y_l^d]$.
- The weight $w_l[C_l, F_l, K_l^{dim}]$ with F_l filters. Each filter has C_l channels and size of K_l^d. In some case, we omit the filter size (also known as kernel size), e.g., $w_l[C_l, F_l]$.
- The activation gradients $\frac{dL}{dy_l}[N, F_l, Y_l^d]$.
- The weight gradients $\frac{dL}{dw_l}[C_l, F_l, K_l^d]$
- The input gradients $\frac{dL}{dx_l}[N, C_l, X_l^d]$.

We adapt non-Conv layers with the above tensors (we extend the notation on [34]). For channel-wise layers, such as pooling or batch-normalization, we require no further adaption. A fully-connected layer with input $x[N, C, W \times H]$ and F output can be expressed as a convolution layer where the size of filters is exactly similar to the size of the input layer, i.e., $w[C, F, W \times H]$, with padding and stride set to 0 and 1, respectively. Thus, the output will become $y[N, F, 1 \times 1]$. For element-wise layers, such as ReLU, the number of filters F is equal to the number of channels C. For layers without weight, such as pooling and ReLU, the weight becomes $w[C, F, 0]$.

The sequential implementation of DNN requires the following steps for each layer :

(IO) $x[N, C, X] \leftarrow IO(\text{dataset, B})$ in the first layer
(FB) $y[N, F, Y] \leftarrow FW(x[N, C, X], w[C, F, K])$
(FB) $\frac{dL}{dx}[N, C, X] \leftarrow BW_{data}(\frac{dL}{dy}[N, F, Y], w[C, F, K])$
(FB) $\frac{dL}{dw}[C, F, K] \leftarrow BW_{weight}(\frac{dL}{dy}[N, F, Y], x[N, C, X])$

TABLE II: Parameters and Notations

D	Data set size
B	Mini batch size
I	Number of iterations per epoch. $I = \frac{D}{B}$
E	Number of epochs
G	Number of layers
x_l	Input of a layer l
y_l	Output (activation) of layer l
w_l	Weight of layer l
W_l / H_l	Width / Height of input of layer l
C_l	Number of input channels of layer l
F_l	Number of output channels of layer l, e.g., number of filters in a convolution layer
FW_l / BW_l	Forward / Backward propagation action of layer l
$[A_1, \ldots, A_n]$	n-dimensions array with size of $A_1 \times A_2 \times \cdots \times A_n$
X_l^d	a d-dimension tuple (array) presents an input channel. In a 2-D convolution layer, X_l^2 is a Cartesian product of $W_l \times H_l$
Y_l^d	a d-dimension output channel
K_l^d	a d-dimension filter. In a 2-D convolution, $K_l^2 = K \times K$
p	Total number of processes elements (PEs)
S	Number of segments in pipeline parallelism
α	Time for sending a message from source to destination
β	Time for injecting one byte of message into network
δ	Number of bytes per item, e.g., input, activation, weight

At the end of each iteration, weights of all layers are updated:
(WU) $w[C, F, K] \leftarrow WU(\frac{dL}{dw}[C, F, K], \rho)$

III. Distributed DNN Training Strategies

Training DNNs using a single PE is computationally expensive. Hence distributed training on HPC systems is common for large models and datasets. In this work, we cover four basic parallel strategies that differ in the way we split the data and model dimensions in the training of DNNs: (1) distributing the data samples among PEs (*data parallelism*), (2) splitting the data sample by its spatial dimension such as width or height (*spatial parallelism*) [37], (3) vertically partitioning the neural network according to its depth (*layer parallelism*) and overlapping computation between one layer and the next layer [3] (also known as *pipeline parallelism*), and (4) horizontally dividing the neural network in each layer by the number of input and/or output channels (*channel and filters parallelism*) [33], [37]. In addition, we name a combination of two (or more) types of the main strategies as *hybrid parallelism* (e.g., data+filter parallelism or *df* for short).

In this work, when presenting the tensors such as x, y, and w, we use the $*$ symbol to present a dimension for which its values are replicated between processes. To emphasize that a tensor's dimension is partitioned among different PEs, we use the number of processes p. For example, in data parallelism, $x[p, *, *]$ implies that the input x is split equally in dimension N (number of samples) and partitioned to p PEs. The other dimensions such as C and X are replicated. The arrow $\xleftarrow{Allreduce}$ presents communication with an Allreduce.

It is important to note that the notation and analysis in this paper is general to input tensors of any dimension (1D, 2D, and 3D). Input tensors of higher dimensions are also valid in our analysis since the input can be represented as 3D tensor with the extra dimensions as component vector(s) (e.g. CosmoFlow [8] has 4D input represented as a 3D tensor plus a vector at each cell). Finally, the parallel strategy can alternatively be viewed as a domain decomposition problem: a recurring problem in HPC applications. Accordingly, we formulate the notation and analysis to be interpretable as domain decomposition schemes.

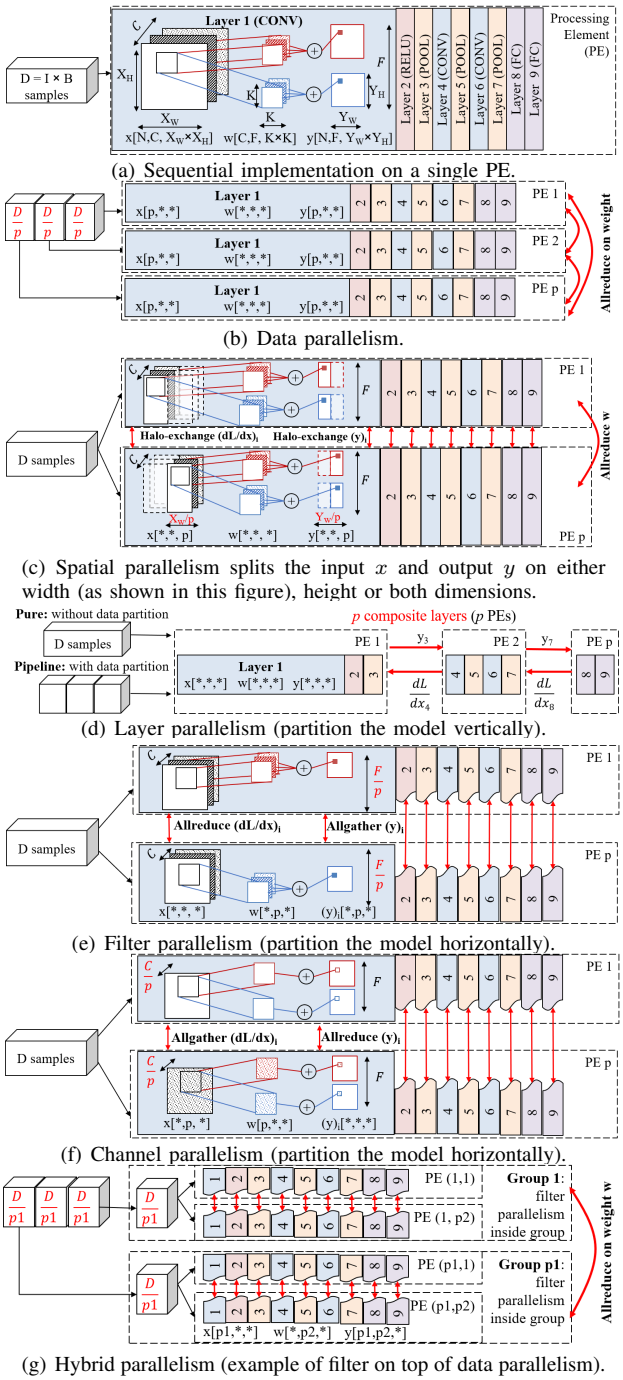

Fig. 1: Different strategies for distributed training of a DNN. Red solid lines refer to communication.

*1) **Data parallelism (sample)**:* The entire model is replicated on p different PEs, e.g., GPUs (Figure 1(b)) and the dataset is scattered into sub-datasets to each PE. Then the forward and backward phases are computed independently, using those different partitions of the dataset, i.e., in a micro-batch $B' = \frac{B}{p}$ at each iteration. In the gradient exchange phase, an Allreduce operation is required to aggregate the

weight gradients, i.e., $\sum_{i=1}^{p}\left(\frac{dL}{dw}\right)_i$. We define operations at the processing element i in data parallelism as:

(IO) $(x)_i[p,*,*] \leftarrow IO(\text{sub-dataset}_i, B')$ in the first layer.
(FB) $(y)_i[p,*,*] \leftarrow FW(x_i[p,*,*], w[*,*,*])$
(FB) $(\frac{dL}{dx})_i[p,*,*] \leftarrow BW_{data}((\frac{dL}{dy})_i[p,*,*], w[*,*,*])$
(FB) $(\frac{dL}{dw})_i[*,*,*] \leftarrow BW_{weight}((\frac{dL}{dy})_i[p,*,*], (x)_i[p,*,*])$
(GE) $\frac{dL}{dw}[*,*,*] \xleftarrow{Allreduce} \sum_{i=1}^{p}\left((\frac{dL}{dw})_i[*,*,*]\right)$
(WU) $w[*,*,*] \leftarrow WU(\frac{dL}{dw}[*,*,*])$

*2) **Spatial parallelism (height-width-deep)**:* All the PEs work on the same batch of samples. First, one leader PE loads those samples at each iteration and then distributes to other PEs. Note that, the spatial dimension H, W (and D as in 3-D convolution layer), of x, y, $\frac{dL}{dx}$ and $\frac{dL}{dy}$ are split among p PEs (Figure 1(c)). That is $p = pw \times ph \times pd$ where pw, ph, $pd \leq W$, H, D, respectively. Each process thus performs the forward and backward operation locally. For a convolution layer, when a filter of size $K \times K$ where $K > 1$ is placed near the border of a partition, each PE requires remote data for computing. Thus, a small number (e.g., $\frac{K}{2}$) of rows and/or columns will be transferred from logically-neighboring remote PEs (halo exchange) [37]. The exchanged data size (i.e., halo(x_l)), depends on how each spatial dimension is split. For example, a processing element i needs a halo exchange for its partial input $(x)_i$ to get $(x)_{i+}$ when computing the output $(y)_i$ in the forward phase. In the backward phase, the computation of $(\frac{dL}{dx})_i$ requires a halo exchange on the corresponding $(\frac{dL}{dy})_i$. To compute the weight gradients requires the $(x)_{i+}$ yet no more halo exchange is required since the exchanged values of $(x)_i$ can be reused. In the weight update phase, an Allreduce is performed to complete the sum of $\frac{dL}{dw}$.

(IO) $x[*,*,*] \leftarrow IO(\text{dataset}, B)$
(IO) $(x)_i[*,*,p] \xleftarrow{Scatter} x[*,*,*]$ in the first layer.
(FB) $(x)_{i+}[*,*,p] \xleftarrow{halo} (x)_i[*,*,p]$
(FB) $(y)_i[*,*,p] \leftarrow FW((x)_{i+}[*,*,p], w[*,*,*])$
(FB) $(\frac{dL}{dy})_{i+}[*,*,p] \xleftarrow{halo} (\frac{dL}{dy})_i[*,*,p]$
(FB) $(\frac{dL}{dx})_i[*,*,p] \leftarrow BW_{data}((\frac{dL}{dy})_{i+}[*,*,p], w[*,*,*])$
(FB) $(\frac{dL}{dw})_i[*,*,*] \leftarrow BW_{weight}((\frac{dL}{dy})_i[*,*,p], (x)_{i+}[*,*,p])$
(GE) $\frac{dL}{dw}[*,*,*] \xleftarrow{Allreduce} \sum_{i=1}^{p}\left((\frac{dL}{dw})_i[*,*,*]\right)$
(WU) $w[*,*,*] \leftarrow WU(\frac{dL}{dw}[*,*,*])$

*3) **Model-vertical (layer) parallelism**:* A model parallel variant at which the DNN is partitioned across its depth (number of layers G) into $p \leq G$ composite layers, where each composite layer is assigned into one PE, as shown in Figure 1(d). We consider the pipeline implementation of this model parallelism (first proposed by GPipe [3]). The mini-batch is divided into S segments of size $\frac{B}{S}$. In each stage, the forward computation of a composite layer i-th on a data segment s is performed simultaneously with the computation of composite layer $(i+1)$-th on the data segment $s-1$ and so on. The backward computation is done in reversed order.

*4) **Model-horizontal parallelism (filter/channel)**:* A model parallel variant in which each layer of the neural network model is equally divided by the number of output (filters F) or input channels (channels C) and distributed on p PEs. Each PE keeps a portion of the weights of a given layer and partially computes the output in both the forward and backward phases. For example, the filter parallelism of a convolution layer [33] is illustrated in Figure 1(e). Each PE i keeps $\frac{F}{p}$ filters and computes $\frac{F}{p}$ corresponding channels of the output activation. That is, $|(y)_i| = N \times |Y| \times \frac{F}{p}$. After finishing the forward computation of each layer, the PEs have to share their local output, i.e., $y = \bigcup_{i=1}^{p}(y)_i$ (via an Allgather operation). After finishing the backward computation of each layer, the processes also have to share their gradient of the input (pass it to the preceding layer), i.e., $\frac{dL}{dx} = \sum_{i=1}^{p}(\frac{dL}{dx})_i$ (an Allreduce operation[1]). Because each PE performs the weight-update on its portion of weights, the gradient-exchange phase is skipped.

(IO) $x[*,*,*] \leftarrow IO(\text{dataset}, B)$
(IO) $(x)_i[*,*,*] \xleftarrow{Bcast} x[*,*,*]$ in the first layer.
(FB) $(y)_i[*,p,*] \leftarrow FW((x)_i[*,*,*], w[*,p,*])$
(FB) $y[*,*,*] \xleftarrow{Allgather} \bigcup_{i=1}^{p}\left((y)_i[*,p,*]\right)$
(FB) $(\frac{dL}{dx})_i[*,*,*] \leftarrow BW_{data}((\frac{dL}{dy})_i[*,p,*], w[*,p,*])$
(FB) $\frac{dL}{dx}[*,*,*] \xleftarrow{Allreduce} \sum_{i=1}^{p}\left((\frac{dL}{dx})_i[*,*,*]\right)\big)$
(FB) $\frac{dL}{dw}[*,p,*] \leftarrow BW_{weight}((\frac{dL}{dy})_i[*,p,*], (x)_i[*,*,*])$
(WU) $w[*,p,*] \leftarrow WU(\frac{dL}{dw}[*,p,*])$

Channel parallelism [34] (Figure 1(f)) is similar to filter parallel strategy but it requires an Allreduce in the forward pass and Allgather in the backward pass.

(FB) $(y)_i[*,*,*] \leftarrow FW((x)_i[*,p,*], w[p,*,*])$
(FB) $y[*,*,*] \xleftarrow{Allreduce} \sum_{i=1}^{p}\left((y)_i[*,*,*]\right)$
(FB) $(\frac{dL}{dx})_i[*,p,*] \leftarrow BW_{data}((\frac{dL}{dy})_i[*,*,*], w[p,*,*])$
(FB) $\frac{dL}{dx}[*,*,*] \xleftarrow{Allgather} \bigcup_{i=1}^{p}\left((\frac{dL}{dx})_i[*,p,*]\right)$

*5) **Hybrid parallelism**:* We have defined four different main parallel strategies which split the dimension N, $W \times H$ ($\times D$), F, C, and G, respectively. Without loss of generalization, a layer also can be split by the size of kernel $K \times K$. However, in practice K is so small that parallelizing by dividing K would not give any benefit. Therefore, we focus on the mentioned main strategies. A hybrid parallelism is a combination of two (or more) strategies. For example, Figure 1(g) illustrates the data+filter parallelism. In which, p PEs are arranged into $p1$ groups of size $p2 = \frac{p}{p1}$. This hybrid strategy implements the filter parallelism inside each group and data parallelism between groups. For a PE $1 \leq i \leq p2$ in a group $1 \leq j \leq p1$:

(IO) $(x)_j[p1,*,*] \leftarrow IO(\text{sub-dataset}_j, B')$
(IO) $(x)_{ij}[p1,*,*] \xleftarrow{Bcast} (x)_j[p1,*,*]$ in the first layer
Filter parallelism inside a group of $p2$ PEs:
(FB) $(y)_{ij}[p1,p2,*] \leftarrow FW((x)_{ij}[p1,*,*], w[*,p2,*])$
(FB) $y_j[p1,*,*] \xleftarrow{Allgather} \bigcup_{i=1}^{p2}\left((y)_{ij}[p1,p2,*]\right)$
(FB) $(\frac{dL}{dx})_{ij}[p1,*,*] \leftarrow BW_{data}((\frac{dL}{dy})_{ij}[p1,p2,*], w[*,p2,*])$
(FB) $(\frac{dL}{dx})_j[p1,*,*] \xleftarrow{Allreduce} \sum_{i=1}^{p2}\left(\frac{dL}{dx})_{ij}[p1,*,*]\right)\big)$
Data parallelism between $p1$ groups :
(FB) $(\frac{dL}{dw})_j[*,p2,*] \leftarrow BW_{weight}((\frac{dL}{dy})_{ij}[p1,p2,*], (x)_{ij}[p1,*,*])$
(GE) $\frac{dL}{dw}[*,p2,*] \xleftarrow{Allreduce} \sum_{j=1}^{p1}\left((\frac{dL}{dw})_j[*,p2,*]\right)$
(WU) $w[*,p2,*] \leftarrow WU(\frac{dL}{dw}[*,p2,*])$

Another example of proposed hybrid parallelism is the combination of data, spatial, filter and channel for convolu-

[1]In the backward phase, because a given layer $l-1$ only requires to use one partition of the layer l's input gradients, i.e., $\frac{dL}{dx}[*,p,*]$, it is possible to perform a Reduce-Scatter instead of an Allreduce operation [34].

tion neural network (CNN) [34]. Further more, the hybrid strategy could be more complex when applying different parallel strategies for different layers [38], [35]. For instance, the parallel strategy of Transformer networks in Megatron-LM [39] partitions a general matrix multiply (GEMM) with filter-wise and then partitions the next GEMM channel-wise.

IV. Model-driven Analysis

In this section, we introduce our analytical model to identify the requirements, limitations and bottlenecks of different parallel strategies when scaling the training of a DNN on a specific system. We quantify and discuss those limitations and bottlenecks via a detailed empirical evaluation in Section V.

A. Assumptions and Restrictions

Targeted models and datasets: our analysis does not cover all possible types of layers used in DNNs, though it can be easily extended to include other new types of layers.

Training time and memory estimation: we assume that all the training data is available in memory before starting the training process. In other words, for this model, the training time does not include the time for I/O. Nonetheless, in our empirical study we discuss a few points on the I/O overhead.

We conservatively estimate the memory required on a per layer basis. For instance, we assume the memory buffers of the output of layer l are different from the memory buffers for the input of layer $l+1$, despite in reality both buffers being the same. There is a variety of optimizations that frameworks implement to reduce the memory used (as shown in table I). Since those optimization methods are complexly intertwined and depend on the framework implementations (sometimes missing), we do not include them in our model.

Parallel strategies: all results in this paper, unless otherwise stated, are for the De Facto scaling approach in DL: weak scaling. The mini-batch size scales with the number of PE, hence making the number of samples per PE remains constant. In addition, unless mentioned, we do not actively optimize for changing the type of parallelism between different layers in a model, i.e., different layers do not have different parallel strategy. However, there can be cases at which a different type of parallelism is used, in order to avoid performance degradation. For instance, the fully connected layer in spatial parallelism is not spatially parallelized, since that would incur high communication overhead for a layer that is typically a fraction of the compute cost of convolution layers [38].

B. Performance and Memory Analysis

In the following, we estimate the total training time in one epoch and maximum memory per PE for the mentioned main parallel strategies and one hybrid strategy. Let FW_l, BW_l denote the time to perform the computation of forward and backward propagation for one sample and let WU_l denote the time for weight update per iteration at layer l [2]. $T_{ar}(p,m)$, $T_{ag}(p,m)$, and $T_{p2p}(m)$ stand for the time of transferring a m-size data buffer between p PEs via an Allreduce, Allgather and in a peer-to-peer scheme. In data parallelism, the training time includes both computation and communication time. Each PE processes a micro batch size $B' = \frac{B}{p}$ in this case. The time for FW and BW in one iteration is $\frac{1}{p}$ of the single-process. Thus the total computation time in one epoch becomes:

$$T_{data,comp} = \frac{D}{p}\sum_{l=1}^{G}(FW_l + BW_l) + \frac{D}{B}\sum_{l=1}^{G}(WU_l) \quad (1)$$

Because PEs have to share their gradients at the end of each iteration, the time for communication is $\frac{D}{B}T_{ar}(p, \sum_{l=1}^{G}|w_l|)$.

Considering the memory footprint, in data parallelism we duplicate the entire model on p different PEs. Each PE processes a partition of the dataset in a microbatch of $B' = \frac{B}{p}$ samples. A layer l mainly needs memory to store its input $B'|x_l|$, activation $B'|y_l|$, weights $|w_i|$, the gradients $B'|\frac{dL}{dx_l}|$, $B'|\frac{dL}{dy_l}|$, and $|\frac{dL}{dw_l}|$. Some models with a huge number of parameters may require significant memory for "bias" such as in a fully-connected layer. We consider to not formally show such memory requirement to make our analysis easy to follow. Overall, if each item of the input, activation, weight and gradients are stored in δ bytes, the maximum required memory at one PE is:

$$M_{data} = \sum_{l=1}^{G}\delta(B'(|x_l| + |y_l|) + |w_l| + B'(|\frac{dL}{dx_l}| + |\frac{dL}{dy_l}|) + |\frac{dL}{dw_l}|) \quad (2)$$

Our analytical model: In theory, the computation time can be estimated by observing the dataset and DNN model (e.g., FLOP counts and the computation speed of each PE). In addition, we model the communication time based on the $\alpha-\beta$ model. The peer-to-peer communication time of transferring a message of size m is modeled by $T_{p2p}(p,m) = \alpha + m\beta$. In which α is the time for a message send from a source to a destination (also known as startup time) and β is the time to inject one byte of data into the network. We follow the common practice in DL communication libraries such as NCCL [40] to use the ring-based algorithm for all the collective communication operation with large message sizes. In this algorithm, a logical ring is first constructed among p PEs based on the system network architecture. Then each PE partitions its m-size data buffer into p segments of size $\frac{m}{p}$. Each PE then sends one data segment to the successive PE and receive another segment from the preceding PE along the ring, i.e., a total of $p-1$ steps for Allgather and $2(p-1)$ steps for Allreduce. Thus $T_{ar}(p,m)$ and $T_{ag}(p,m)$ can be modeled by $2(p-1)(\alpha + \frac{m}{p}\beta)$ and $(p-1)(\alpha + m\beta)$, respectively. Based on this communication model, we also estimate the total training time in one epoch and maximum memory per PE for the mentioned parallel strategies. We summarize our analytical model in Table III. We present the details of this analysis in our APPENDIX.

The interconnect hierarchy of modern computing systems and the communication technologies (such as GPUDirect [41]) may affect the accuracy of communication prediction due to the difference of latency and bandwidth factors α and β at different interconnect levels. In this work, the start up time α of a given pair is estimated as the total switching latency, which

[2] In pipeline, each PE i keeps G_i layers of the model given that $\sum_{i=1}^{p} G_i = G$. Let FW_{G_i}, BW_{G_i} and WU_{G_i} denote the time for performing the forward, backward, and weight update computation of group i per sample.

TABLE III: Computation, Communication, and Memory Analysis Summary (per epoch)

	Computation Time T_{comp}	**Communication Time T_{comm}**	**Maximum Memory Per PE**	**Number of PEs p**
Serial	$D\sum_{l=1}^{G}\Big(FW_l + BW_l\Big) + \frac{D}{B}\sum_{l=1}^{G}(WU_l)$	0	$2\delta\sum_{l=1}^{G}\Big(B(\lvert x_l\rvert + \lvert y_l\rvert) + \lvert w_l\rvert\Big)$	$p = 1$
Data	$\frac{D}{p}\sum_{l=1}^{G}(FW_l + BW_l) + \frac{D}{B}\sum_{l=1}^{G}(WU_l)$	$2\frac{D}{B}(p-1)\Big(\alpha + \frac{\sum_{l=1}^{G}\lvert w_l\rvert}{p}\delta\beta\Big)$	$2\delta\sum_{l=1}^{G}\Big(\frac{B}{p}(\lvert x_l\rvert + \lvert y_l\rvert) + \lvert w_l\rvert\Big)$	$p \leq B$
Spatial	$\frac{D}{p}\sum_{l=1}^{G}\Big(FW_l + BW_l\Big) + \frac{D}{B}\sum_{l=1}^{G}(WU_l)$	$2\frac{D}{B}\Big((p-1)(\alpha + \frac{\sum_{l=1}^{G}\lvert w_l\rvert}{p}\delta\beta) + \sum_{l=1}^{G}\big(2\alpha + B(\text{halo}(\lvert x_l\rvert) + \text{halo}(\lvert\frac{dL}{dy_l}\rvert))\delta\beta\big)\Big)$	$2\delta\sum_{l=1}^{G}\Big(B\frac{(\lvert x_l\rvert+\lvert y_l\rvert)}{p} + \lvert w_l\rvert\Big)$	$p = pw \times ph \leq \min_{l=1}^{G}(W_l \times H_l)$
Layer (Pipeline)	$\frac{D(p+S-1)}{S}\Big(\max_{i=1}^{p}(FW_{G_i}) + \max_{i=1}^{p}(BW_{G_i})\Big) + \max_{i=1}^{p}(WU_{G_i})\Big)$	$2\frac{D(p+S-2)}{B}\Big(\max_{i=1}^{p-1}\big(\alpha + \frac{B}{S}\lvert y_{G_i}\rvert\delta\beta\big)\Big)$	$2\delta\max_{i=1}^{p}\Big(\sum_{l=1}^{G_i}\big(B(\lvert x_l\rvert + \lvert y_l\rvert) + \lvert w_l\rvert\big)\Big)$	$p \leq G$
Filter	$\frac{D}{p}\sum_{l=1}^{G}\Big(FW_l + BW_l\Big) + \frac{D}{Bp}\sum_{l=1}^{G}(WU_l)$	$3\frac{D}{B}(p-1)\sum_{l=1}^{G-1}(\alpha + \frac{B\lvert y_l\rvert}{p}\delta\beta)$	$2\delta\sum_{l=1}^{G}\Big(B(\lvert x_l\rvert + \lvert y_l\rvert) + \frac{\lvert w_l\rvert}{p}\Big)$	$p \leq \min_{l=1}^{G}(F_l)$
Channel	$\frac{D}{p}\sum_{l=1}^{G}\Big(FW_l + BW_l\Big) + \frac{D}{Bp}\sum_{l=1}^{G}(WU_l)$	$3\frac{D}{B}(p-1)\sum_{l=1}^{G-1}(\alpha + \frac{B\lvert y_l\rvert}{p}\delta\beta)$	$2\delta\sum_{l=1}^{G}\Big(B(\lvert x_l\rvert + \lvert y_l\rvert) + \frac{\lvert w_l\rvert}{p}\Big)$	$p \leq \min_{l=1}^{G}(C_l)$
Data + Filter	$\frac{D}{p}\sum_{l=1}^{G}\Big(FW_l + BW_l\Big) + \frac{D}{Bp2}\sum_{l=1}^{G}(WU_l)$	$3\frac{D}{B}(p2-1)\sum_{l=1}^{G-1}(\alpha + \frac{B\lvert y_l\rvert}{p}\delta\beta) + 2\frac{D}{B}(p1-1)(\alpha + \frac{\sum_{l=1}^{G}\lvert w_l\rvert}{p}\delta\beta)$	$2\delta\sum_{l=1}^{G}\Big(\frac{B(\lvert x_l\rvert+\lvert y_l\rvert)}{p1} + \frac{\lvert w_l\rvert}{p2}\Big)$	$p = p1 \times p2 \leq B \times \min_{l=1}^{G}(F_l)$

TABLE IV: Models and Datasets Used in Experiments. *tz*: token size.

Model	Dataset	#Samples (Size)	# Param.	#Layers
ResNet-50 [45]	ImageNet [18]	1.28M (3×226^2)	$\approx$ 25M	50
ResNet-152 [45]			$\approx$ 58M	152
VGG16 [46]			$\approx$ 169M	38
CosmoFlow [8]	CosmoFlow [47]	1584 (4×512^3)	$\approx$ 2M	20
Megatron-LM [39]	OpenWT [48]	7.2M (1024$\times$ *tz*)	$\approx$ 2.5B	54

depends on the number of intermediate switching elements. In addition, β is the inverse of the minimum link bandwidth on the routing path between two PEs (the bottleneck link). It is important to repeat that collective algorithms can be broken down into many steps in a synchronous manner. Thus, the communication time of each step is estimated as the maximum communication time of all the PE pairs along the ring.

It is important to note that processors, CPU or GPU, rarely perform close to their peak. We empirically profile the average computation time per sample of each layer (or group of layers) on the target architecture to get a more accurate result. Such profiling can be performed easily and quickly beforehand. Further, the empirical compute time, per a given layer on a given processor, is available in DL databases, such as the one proposed by Li et al. [42]. Similarly, we can also use the empirical measurements of running MPI benchmarks on a specific computing system as a model for communication time [43]. we use empirical measurements to get the total computation time per sample in the sequential strategies.

V. Evaluation

In this section, we conduct a wide range of experiments to show the utility of the analytical model in projecting the ideal performance of training DL models under different parallel strategies. We compare the theoretical results to the empirical measurements on a real HPC system, and characterize the bottlenecks when the measured values are far from the projected ones. Note that in this section, we don't focus on identifying the highest performing parallel strategy; instead we identify limitations and bottlenecks at scale, and discuss those findings.

A. Methodology

Selected Models and Datasets: We choose different types of DNN models and datasets with different characteristics and features which affect both performance and memory requirements (summarized in Table IV) as test cases.

Evaluation Environment: Experiments are performed on ABCI supercomputer, in which each compute node has two Intel Xeon Gold 6148 Processors and four NVIDIA Tesla V100 GPUs (16GB of memory per GPU). The GPUs are connected intra-node to the CPUs by PLX switches and PCIe Gen3 x16 links, and together by NVLink. The compute nodes are connected in a 3-level fat-tree topology which has full-bisection bandwidth, and 1:3 over-subscription for intra-rack and inter-rack, respectively (17 compute nodes per rack). Thus, to calculate the ideal communication time, we approximate the leaf, spine, and PLX switching latency to 90, 400, and 110 nanosecond per switch, respectively. We pick those values from the latency of actual Infiniband switches (same as in [12]). The GPU-to-GPU communication startup latency via NVLINK is approximately 20μs as it is estimated in [43]. We also set the link bandwidths to 12.5, 16, and 50 GBps for inter-node, PCIe, and NVLink links, respectively.

Implementation Details: we implement most of the parallel strategies using Chainer v7.0.0 with CUDA v10.0 and ChainerMN. We also reuse the PyTorch implementation for pipeline strategy [44] and Megatron-LM [29]. More specifically, we use the default implementation of Chainer for data parallelism. Since the size of each dimension (i.e., H, W, and/or D) limits the parallelism of spatial strategy, in this work, we implement the spatial strategy for some first layers of a given model until adequate parallelism is exposed while still maintaining the maximum required memory per node within memory capacity. We then implement an Allgather to collect the full set of activations before passing it to the following layers which perform similar to the sequential implementation. For example, we aggregate after the second convolution layer in CosmoFlow because most of required memory footprint and compute are in those first two layers.

For hybrid strategies such as data plus filter (or data plus spatial), we map the data parallel part on inter-node. This implementation is also used in [34]. We leverage ChainerMN with MPI support for both inter-node and intra-node communication. We implement all communication functions based on Nvidia NCCL library [40] (version 2.4.8.1), except for halo exchange of spatial strategy because P2P communication interfaces are not supported by NCCL library.

Configurations of Experiments: An important performance factor is efficient device occupancy of GPUs. Thus, we conducted a series of test runs for each type of parallelism and DL model to identify the optimal number of samples per GPU

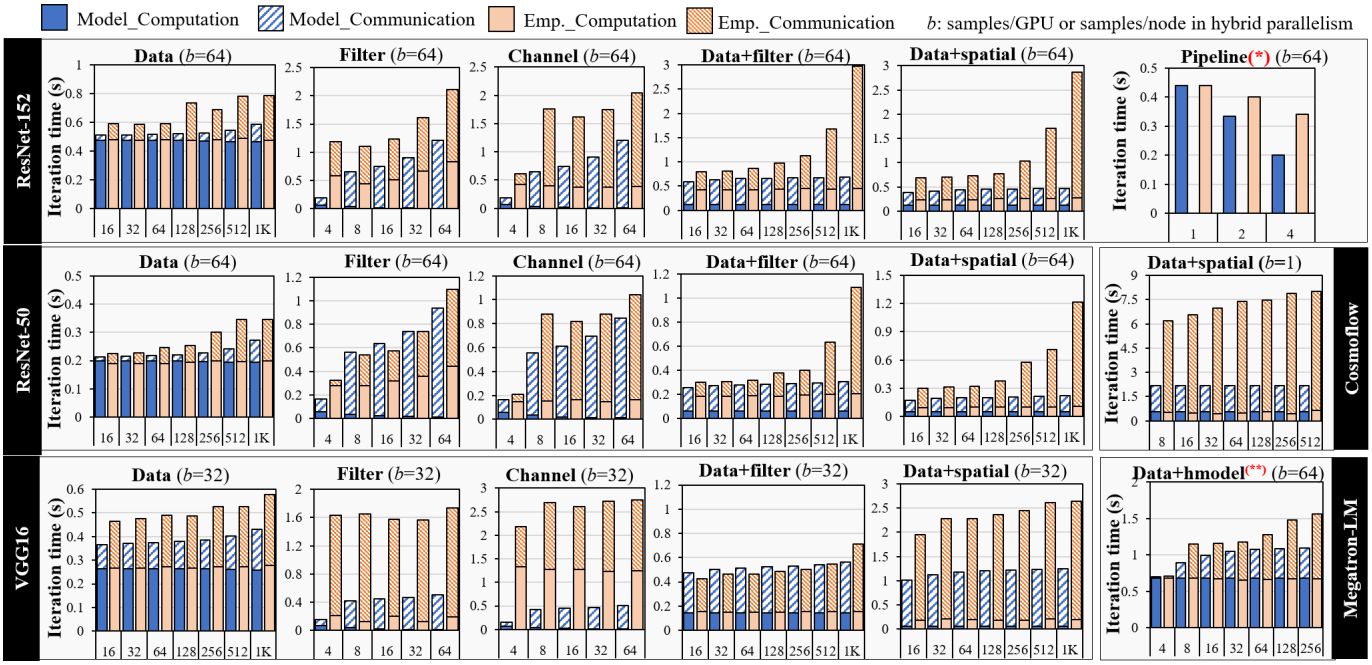

Fig. 2: Time breakdown of our analytical model (Model) in comparison with measured runs (Emp.). The x-axis is the number of GPUs. Filter/channel/pipeline are strong scaling.(*)Values are total time since pipeline parallelism [44] overlaps the computation and communication. (**)The hmodel parallelism in Megatron-LM partitions a GEMM with filter-wise and then partitions the next GEMM channel-wise.

(or node) that would efficiently utilize the device (marked as b in Figure 2). We observed that the performance drops significantly when we train using a higher samples/GPU number than the optimal one. This occurs when the computational load becomes too large to effectively utilize a single GPU. For instance, in CosmoFlow with spatial strategies, although we use only one sample per node, i.e., 0.25 samples/GPU, we could not avoid the performance drop.

B. Limitations of the Parallel Strategies and Bottlenecks

We first compare the projected ideal training time of our analytical model (Model) with empirical results (Emp.) in Figure 2. We then use the combination of observations from model projections and empirical results to highlight some concerning points: (i) inherent to parallel strategies themselves (limitations), and (ii) those caused by other components such as the framework implementation or system architecture (bottlenecks). Table V summarizes the issues detected by a combination of the analytical model and empirically, grouped into four categories.

*1) **Communication**:* It is well-known in literature that parallel training introduces redundant communication that is non-trivial (as in Figure2). Those overheads have different forms and patterns for different parallel strategies. There is the commonly known gradient exchange at the end of each iteration in data and spatial (Allreduce). There can also be extra communication in the forward and backward passes of other parallel strategies: the layer-wise collective communication in filter/channel (FB-Allgather and FB-Allreduce), layer-to-layer communication in pipeline (FB-layer), and the halo exchange in spatial (FB-Halo). We show the breakdown of communication time for different parallel strategies in Figure 3 (measured empirically, and independently verified by the analytical model).

TABLE V: Summary of detected limitations (**L**) and bottlenecks (**B**) with the related training phases (✓) and components (●). Those limitations/bottlenecks may appear (Y/N) in large scale distributed inference (**I**). Related parallel strategies (×): **d**-data, **s**-spatial, **p**-pipeline, **f/c**-filter and channel, **df**-hybrid of data and filter, **ds**-hybrid of data and spatial. **FR**-Framework, **SY**-System.

Category	L/B	Para. Strategies						FR	SY	Training Phases				I	Remarks
		d	s	p	f/c	df	ds			IO	FB	GE	WU		
Commu-nication	L	×	×	-	-	×	×	○	○	-	-	✓	-	N	Gradient-exchange
	L	-	-	-	×	×	-	○	○	-	✓	-	-	Y	Layer-wise comm.
	B	-	×	×	-	-	×	●	○	-	✓	✓	-	Y	P2P communication
	B	×	×	×	×	×	×	○	●	-	✓	✓	-	Y	Network Congestion
	B	-	×	×	×	×	×	●	○	✓	-	-	-	Y	I/O and Staging
Memory Capacity	B	×	×	×	×	×	×	○	●	✓	✓	✓	✓	Y	Memory Redundancy
	B	×	×	×	×	×	×	●	○	✓	✓	✓	✓	Y	Fragmentation
Comput-ation	L	×	×	×	×	×	×	○	○	-	-	-	✓	N	Weight Update
	L	-	-	×	-	-	-	○	○	-	✓	-	✓	Y	Workload Balancing
	B	-	-	-	×	×	-	●	○	-	✓	-	✓	Y	Comp. Redundancy
Scaling	L	×	×	×	×	×	×	○	○	✓	✓	✓	✓	Y	Number of PEs

Gradient Exchange: Similar to data parallelism, spatial requires a gradient exchange to aggregate the weights. This collective communication, i.e., Allreduce, has significant impact on performance, and can become a limitation (as discuss in detail in Section I). Another point worth mentioning is the hierarchal implementation of Allreduce in hybrid strategies such as data plus filter (*df*) and data plus spatial (*ds*). These two types of Allreduce (data and hybrid parallelism) are different as they involve different parallelization techniques. In hybrid, we perform a reduce inside each node to leader GPUs first, then perform a global Allreduce between the leaders. However such implementation leads to a higher overhead, e.g., time of *df* are more than 2× as those of data. Alternative

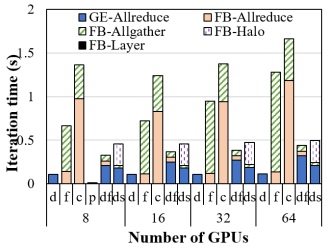

Fig. 3: Communication overhead of ResNet152 with parallel strategies: d-data, f-filter, c:channel, df-(data+filter), ds-(data+spatial).

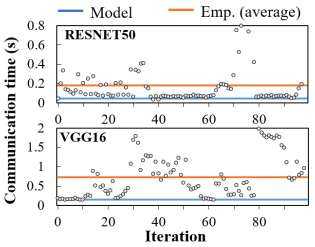

Fig. 4: Network congestion of ResNet50, 512 GPUs, data parallelism (upper) and VGG16, 64 GPUs, filter parallelism (lower).

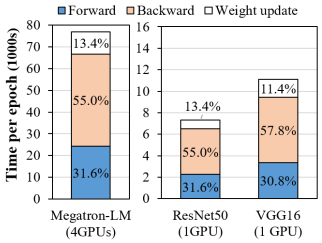

Fig. 5: Computation time per epoch with PyTorch. Weight update is not trivial in large models and dataset.

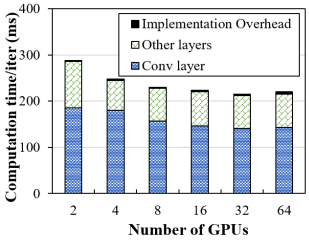

Fig. 6: Computation breakdown of filter parallelism, ResNet50. Implementation of convolution layers does not scale well.

ways to address this issue are to use multiple leaders instead of only one [12] or to use segmented allreduces, i.e., smaller, concurrent allreduces among disjoint sets of GPUs [34].

Layer-wise collective communication: unlike data parallelism, filter and channel parallelism require multiple collective communication rounds at each layer. The communication time depends on the activation size $\times$ batch size, i.e., $\mathcal{O}(B\sum_{l=1}^{G}|y_l|)$, as well as the depth of DL model, i.e., $\sum_{l=1}^{G}(p-1)\alpha$ as reported in our analytic model. In our experiments with ImageNet, even though the total activation sizes are smaller than the number of weights, yet with a batch size of ≥ 32 samples, the communication time of filter/channel are much larger than that of data parallelism (Figure 3). Note that because this communication overhead is attributed to the forward and backward phases, Allreduce optimization techniques mentioned in Section I such as sparsification are no-longer valid. Instead, a hybrid which combines filter and channel (plus data) parallelism may help to mitigate this limitation by reducing the number of communication calls [39] or using a smaller segmented Reduce-Scatter [34].

Peer to Peer communication: The halo exchange in spatial parallelism and the activation passing between composite layers in pipeline are performed in a peer to peer fashion. Thus, our analytical model expects those communication to be small. However, as in Figure 3, the time of FB-Halo is non-trivial. This bottleneck appears because the framework uses the MPI library instead of NCCL (NCCL allows GPUs to communicate directly instead of via CPU, i.e., GPUDirect).

Network Congestion: In our empirical experiment, we try to avoid the issue of congestion as much as possible by running several times for each data point. However, as shown in Figure 2, we still observe network congestion when approaching 1K GPUs. We did a detailed analysis for several of the runs. In Figure 4 we show the time for Allreduce communication for data parallelism of ResNet50 with 512 GPUs and an Allgather communication for filter parallelism of VGG16 with 64 GPUs. We noticed that most data points align well with the expected theoretical bandwidth predicted by our analytical model, yet network congestion generated by other jobs in the system can lead to some outliers that push the average communication time up to four times higher than expected. This overhead can not be avoided especially for large-scale training, e.g., 100s-1000s GPU, in a shared HPC system. It could, however, be mitigated at the system level by switching to a full-bisection bandwidth rather than having 1:3 over-subscription.

*2) **Memory Capacity**:* We highlight specific cases when memory requirements become an issue in different strategies.

Redundancy in Memory: Different memory redundancies could emerge in different parallel strategies. In the spatial and channel/filter strategies the activations (i.e. input/output channels) are divided among nodes, however this does not reduce the memory requirements of holding the weight tensors since their weights are not divided among PEs (as in our analytical model). This becomes an issue for larger models, and hence we could not, for instance, increase the model size we used in our experiments with Meagtron-LM (Table IV). One alternative proposed in ZeRO [6] is to split the weights as well as the activations. However, this comes at the cost of extra communication of 50% since two Allgathers of the weights are needed in the forward and backward pass.

In pipeline, the memory required for a single layer could be prohibitive. For example, ComsoFlow's first Conv layer generates more than 10GB of activation tensor. Accordingly for those kind of models the pipeline strategy would be unfeasible and one has to resort to other parallel strategies (e.g., use spatial+data hybrid for CosmoFlow as shown in Figure 7). Additionally, since samples are feed in a pipelined fashion, the memory required is proportional to the number of stages, and would hence become a bottleneck in deep pipelines, unless we apply gradient checkpointing at the boundary of the partition [3], [49], which comes with the overhead of recomputing the activations within each partition.

Fragmentation: We frequently observed Out-Of-Memory cases, even when the memory manager API reports free memory. More specifically, we observed those cases with Megatron-LM variants with over 1.8B parameters in almost 15% of our submitted runs. Upon inspection, it became clear that the device memory is fragmented during training mainly due to variations in the lifetime of different tensors. One partial solution would be managing memory based on the lifetime of tensors [6], or planning the entire memory allocation plan ahead of the actual running.

*3) **Computation**:* We highlight the following limitations. *Weight update*: most compute time in training typically go

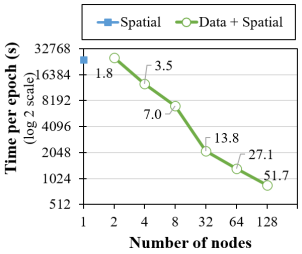

Fig. 7: Spatial + data scaling with CosmoFlow. The labels show the speedup ratio of spatial+data over the pure spatial strategy.

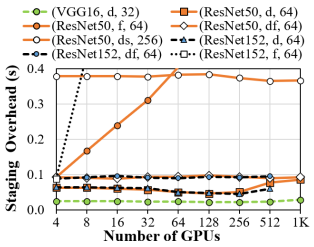

Fig. 8: Staging overhead. Labels show (model, parallel strategy, samples/GPU). d:data, f:filter, df:(data+filter), ds:(data+spatial).

to the forward and backward pass. However, we observed with our analytical model that for larger models the weight update starts to become a significant portion of the compute time. For instance, we measured weight update to take up to 15% for the relatively small configuration (2.5B parameters) of Megatron-LM models (shown in Figure 5). Large Transformer-based models, using ADAM optimizer in specific, report up to 45% time on weight update and more than 60% extra memory requirements since ADAM requires four variables per weight. One alternative to address this is to shard the weight update among GPUs across iterations, and Allgather the weights before forward/backward passes [50].

Workload Balancing: Pipeline strategies can outperform data parallelism with less communication by using peer-to-peer rather than a collective communication. However, it is crucial that all stages in the pipeline take roughly the same amount of time, since the training time of a pipeline is limited by the slowest stage.Indeed, there may be cases where no simple partitioning across the GPUs achieves perfect load balance, hence it restricts the application of pipeline. To further improve load balancing, a straight forward approach is to use data parallelism inside a stage, i.e., hybrid of pipeline plus data [49].

Computation Redundancy: This section discusses limits that could arise from computational redundancy that is introduced for different parallel strategies. In figure 2, the computation time of Model and Emp. fit well for data parallelism because data parallelism splits the dataset and does not change the computation logic. For filter/channel/spatial (and hybrid), the computation time are expected to shrink linearly, to the number of partitions (GPUs) as in our theoretical analysis. However, in practice, our implementation could not achieve the ideal scaling because the computation of a convolution layer barely changes as shown in Figure 6.

*4) **Scaling limitation**:* When scaling, there exists a limit on the number of GPUs for each of the model parallel strategies (last column of Table III). For example, p can not exceed the minimum number of filters of a layer in the model, i.e., 64 in the case of VGG and ResNet50 with the filter parallel strategy. Hybrid approaches have a better scaling than those of pure model parallelisms. For example, as shown in Figure 7, using spatial+data hybrid is an effective scalable alternative (despite communication inefficiencies), since it scales both in performance and GPU count (i.e. one could simply expand the data parallel pool as much as new nodes are added). Indeed, the curve shows a perfect scaling (note the logarithmic y-axis).

C. Other Observations

This section briefly discusses further notable points related to distributed DNNs.

1) Limitations and Bottlenecks in Distributed Inference: Inference at large scale is becoming increasingly demanded, given that for large models the inference will also be distributed [7]. In smaller models, when latency of inference matters in an application, the inference could also be distributed (e.g., real-time prediction of Tokamak disruptions in magnetically-confined thermonuclear plasma experiments [51]). We highlight some limitations and bottleneck of training that also may appear in distributed inference (marked with **Y** in column **I** of table V).

2) Staging and I/O Overhead: Figure 8 shows the staging overhead per iteration of different cases of (DL model, parallel strategy). Because each case loads different number of samples/GPU, we expect to see differences between ResNet50 and VGG16 (64 vs. 32 samples) and a similar difference between ResNet50 and ResNet152 (both use 64 samples). We also noticed that spatial and model (e.g., filter) parallelism load all the samples on a leader process, which then distributes (part of) the samples to the other GPUs. This staging process evidently adds a significant amount of time, especially in a large-scale run or when the sample sizes are large. For example, I/O time is 4× of communication and computation times in the case of CosmoFlow. Note that this is not a fundamental limitation of spatial/model parallelism, but simply an implementation issue of the underlying frameworks.

3) The Rise of Hybrid Parallelism: As mentioned, each of four basic parallelism strategies has its own limitations. Using the hybrid strategies (data plus model) helps to break/mitigate those limitations, e.g., memory issue of data parallelism and communication and scaling limitation of model parallelism (Section V-B). As more datasets from HPC start to be analyzed by DL frameworks, this type of hybrid parallel strategies will become more and more relevant because data parallelism will simply be not enough. In addition, hybrid strategies may drive to a better solution in terms of performance. In accordance with other recent reports [34], there can be cases where data+filter (*df*) hybrid can outperform data parallelism at large scale. In our experiments scaling ResNet50, we can see, at a global batch size of 16K, *df* is 8% faster than data parallelism.

D. Summary of our Analysis

We list the main findings here (Table V shows a compact version of these findings):

- Our analytical model projects ideal performance that gives consistent accuracy on a wide range of models for up to 1K GPUs that we use to identify limitations and bottlenecks.
- We analytically and empirically identify different communication bottlenecks that can appear in different parallel

strategies due to communication patterns that compensate for different ways to split the tensors.
- We identify memory and computational pressure that arise from different redundancies in different parallel strategies.

VI. Conclusion

We propose an analytical model for characterizing and identifying the limitations and bottlenecks of different parallel strategies for DL training. We run a wide range of experiments with different models, different parallel strategies and different datasets for up to 1,000s of GPUs and compare with our analytical model. Based on this, we project the limitation and bottlenecks when the measured values are far from the analysis result. While most works in the literature focus on improving performance for a specific strategy, our analytical model functions as the basis for a utility that aids end-users, framework developers and system builders in identifying the communication, memory, and computations limits and bottlenecks when scaling different parallel strategies.

References

[1] T. Ben-Nun *et al.*, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *CoRR*, vol. abs/1802.09941, 2018.

[2] Y. You *et al.*, "ImageNet Training in Minutes," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018, 2018, pp. 1:1–1:10.

[3] Y. Huang *et al.*, "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism," *CoRR*, vol. abs/1811.06965, 2018.

[4] T. Kurth *et al.*, "Exascale Deep Learning for Climate Analytics," ser. SC '18, 2018, pp. 51:1–51:12.

[5] J. Dean *et al.*, "Large scale distributed deep networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems*, 2012, pp. 1223–1231.

[6] S. Rajbhandari *et al.*, "ZeRO: Memory Optimization Towards Training A Trillion Parameter Models," *ArXiv*, vol. abs/1910.02054, 2019.

[7] T. B. Brown *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[8] A. Mathuriya *et al.*, "Cosmoflow: Using deep learning to learn the universe at scale," ser. SC '18, 2018, pp. 65:1–65:11.

[9] X. Jia *et al.*, "Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes," *CoRR*, vol. abs/1807.11205, 2018.

[10] M. Yamazaki *et al.*, "Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds," *CoRR*, vol. abs/1903.12650, 2019.

[11] M. Bayatpour *et al.*, "Scalable reduction collectives with data partitioning-based multi-leader design," ser. SC '17, 2017, pp. 64:1–64:11.

[12] T. T. Nguyen *et al.*, "Hierarchical Distributed-Memory Multi-Leader MPI-Allreduce for Deep Learning Workloads," ser. CANDAR18, 2018, pp. 216–222.

[13] A. Radford *et al.*, "Language models are unsupervised multitask learners," *OpenAI Tech Report*, 2019.

[14] F. Seide *et al.*, "On parallelizability of stochastic gradient descent for speech DNNS," in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2014, pp. 235–239.

[15] P. Goyal *et al.*, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour," *CoRR*, vol. abs/1706.02677, 2017.

[16] S. L. Smith *et al.*, "Don't Decay the Learning Rate, Increase the Batch Size," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Conference Track Proceedings*, 2018.

[17] T. Akiba *et al.*, "Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes," *CoRR*, vol. abs/1711.04325, 2017.

[18] O. Russakovsky *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[19] B. Ginsburg *et al.*, "Large batch training of convolutional networks with layer-wise adaptive rate scaling," 2018, [01 April 2020]. [Online]. Available: https://openreview.net/forum?id=rJ4uaX2aW

[20] Y. You *et al.*, "Large Batch Optimization for Deep Learning: Training BERT in 76 minutes," 2020, [01 April 2020]. [Online]. Available: https://openreview.net/forum?id=Syx4wnEtvH

[21] J. G. Pauloski *et al.*, "Convolutional neural network training with distributed K-FAC," *arXiv preprint arXiv:2007.00784*, 2020.

[22] H. Zhang *et al.*, "Context encoding for semantic segmentation," in *CVPR2018*, June 2018.

[23] Y. Wu and K. He, "Group normalization," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 3–19.

[24] A. Kolesnikov *et al.*, "Big Transfer (BiT): General Visual Representation Learning," *arXiv preprint arXiv:1912.11370*, 2019.

[25] S. Lym *et al.*, "PruneTrain: Fast Neural Network Training by Dynamic Sparse Model Reconfiguration," ser. SC '19, 2019.

[26] C. Renggli *et al.*, "SparCML: High-Performance Sparse Communication for Machine Learning," ser. SC '19, 2019.

[27] F. Seide *et al.*, "1-Bit Stochastic Gradient Descent and Application to Data-Parallel Distributed Training of Speech DNNs," ser. Interspeech 2014, September 2014.

[28] T. Chen *et al.*, "Training Deep Nets with Sublinear Memory Cost," *ArXiv*, vol. abs/1604.06174, 2016.

[29] M. Wahib *et al.*, "Scaling Distributed Deep Learning Workloads beyond the Memory Capacity with KARMA," *arXiv preprint arXiv:2008.11421*, 2020.

[30] J. Domke *et al.*, "HyperX Topology: First at-Scale Implementation and Comparison to the Fat-Tree," ser. SC '19, 2019.

[31] J. Dong *et al.*, "EFLOPS: Algorithm and System Co-Design for a High Performance Distributed Training Platform," in *HPCA*, 2020, pp. 610–622.

[32] A. Castelló *et al.*, "Analysis of Model Parallelism for Distributed Neural Networks," ser. EuroMPI '19, 2019, pp. 7:1–7:10.

[33] A. Gholami *et al.*, "Integrated model, batch and domain parallelism in training neural networks," *arXiv preprint arXiv:1712.04432*, 2017.

[34] N. Dryden *et al.*, "Channel and Filter Parallelism for Large-Scale CNN Training," ser. SC '19, 2019, pp. 46:1–46:13.

[35] Z. Jia *et al.*, "Exploring hidden dimensions in parallelizing convolutional neural networks," *arXiv preprint arXiv:1802.04924*, 2018.

[36] J. Zhihao *et al.*, "Beyond Data and Model Parallelism for Deep Neural Networks," *CoRR*, vol. abs/1807.05358, 2018.

[37] N. Dryden *et al.*, "Improving Strong-Scaling of CNN Training by Exploiting Finer-Grained Parallelism," in *IPDPS 2019*, pp. 210–220.

[38] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[39] M. Shoeybi *et al.*, "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism," *ArXiv*, vol. abs/1909.08053, 2019.

[40] "NVIDIA Collective Communications Library (NCCL), Multi-GPU and multi-node collective communication primitives," https://developer.nvidia.com/nccl, [01 April 2020].

[41] "GPUDirect," https://developer.nvidia.com/gpudirect, [21 April 2020].

[42] C. Li *et al.*, "Benanza: Automatic Benchmark Generation to Compute "Lower-bound" Latency and Inform Optimizations of Deep Learning Models on GPUs," *ArXiv*, vol. abs/1911.06922, 2019.

[43] A. Li *et al.*, "Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect," *CoRR*, vol. abs/1903.04611, 2019.

[44] C. Kim *et al.*, "torchgpipe: On-the-fly pipeline parallelism for training giant models," *arXiv preprint arXiv:2004.09910*, 2020.

[45] K. He *et al.*, "Deep residual learning for image recognition," in *CVPR*, 2016, pp. 770–778.

[46] K. Simonyan *et al.*, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *ICLR 2015*, 2015.

[47] National Energy Research Scientific Computing Center, "CosmoFlow datasets," https://portal.nersc.gov/project/m3363/, [15 January 2020].

[48] J. Peterson and other, "OpenWebText Dataset," https://github.com/jcpeterson/openwebtext, [22 February 2020].

[49] D. Narayanan *et al.*, "PipeDream: Generalized Pipeline Parallelism for DNN Training," ser. SOSP '19, 2019, p. 115.

[50] Y. Xu *et al.*, "Automatic Cross-Replica Sharding of Weight Update in Data-Parallel Training," *arXiv preprint arXiv:2004.13336*, 2020.

[51] G. Dong *et al.*, "Fully convolutional spatio-temporal models for representation learning in plasma science," *arXiv preprint arXiv:2007.10468*, 2020.

[52] S. Tokui *et al.*, "Chainer: a Next-Generation Open Source Framework for Deep Learning," in *Proceedings of LearningSys in NIPS*, 2015.

[53] T. Akiba *et al.*, "ChainerMN: Scalable Distributed Deep Learning Framework," in *Workshop on ML Systems in NIPS*, 2017.
[54] I. Sergey *et al.*, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *CoRR*, vol. abs/1502.03167, 2015.
[55] Y. Wang *et al.*, "Eidetic 3D LSTM: A model for video prediction and beyond," 2018.

VII. Appendix

A. Detail of Performance and Memory Analysis

The training time of one epoch in the sequential implementation (serial) of DNN includes only the time for computation:

$$\begin{aligned} T_{serial} &= \sum_{1}^{I} B \sum_{l=1}^{G} \Big(FW_l + BW_l \Big) + \sum_{1}^{I} \sum_{l=1}^{G} \Big(WU_l \Big) \\ &= D \sum_{l=1}^{G} \Big(FW_l + BW_l \Big) + \frac{D}{B} \sum_{l=1}^{G} (WU_l) \end{aligned} \tag{3}$$

Considering the memory footprint:

$$M_{serial} = 2\delta \sum_{l=1}^{G} \Big(B(|x_l| + |y_l|) + |w_l| \Big) \tag{4}$$

In the following, we estimate the total training time and maximum memory per PE for the mentioned basic parallelism strategies and one hybrid strategy.

1) Data parallelism: In this strategy, the training time includes both computation and communication time. Each PE processes a micro batch size $B' = \frac{B}{p}$ in this case. The time for computing at layer l in one iteration for forward and backward phase is $\frac{1}{p}$ of the single-process. Thus the total computation time in one epoch becomes:

$$\begin{aligned} T_{data,comp} &= \sum_{1}^{I} \sum_{l=1}^{G} \Big(\frac{B}{p} (FW_l + BW_l) + WU_l \Big) \\ &= \frac{D}{p} \sum_{l=1}^{G} (FW_l + BW_l) + \frac{D}{B} \sum_{l=1}^{G} (WU_l) \end{aligned} \tag{5}$$

Because PEs have to share their gradients at the end of each iteration, the time for communication is $\frac{D}{B} T_{ar}(p, \sum_{l=1}^{G} |w_l|)$. i.e., an Allreduce operation with a ring-based algorithm, the time for communication is:

$$T_{data,comm} = 2\frac{D}{B}(p-1)\Big(\alpha + \frac{\sum_{l=1}^{G} |w_l|}{p} \delta\beta \Big) \tag{6}$$

Clearly, data parallelism has the benefit of reduction in computation time by $\frac{1}{p}$ at the price of communication time.

Considering the memory footprint, in data parallelism we duplicate the entire model on p different PEs. Each PE processes a partition of the dataset in a microbatch of $B' = \frac{B}{p}$ samples. A layer l mainly needs memory to store its input $B'|x_l|$, activation $B'|y_l|$, weights $|w_i|$, the gradients $B'|\frac{dL}{dx_l}|$, $B'|\frac{dL}{dy_l}|$, and $|\frac{dL}{dw_l}|$. Some models with a huge number of parameters may require significant memory for "bias" such as in a fully-connected layer. We consider to not formally show such memory requirement to make our analysis easy to follow. Overall, if each item of the input, activation, weight and gradients are stored in δ bytes, the maximum required memory at one PE is:

$$\begin{aligned} M_{data} &= \sum_{l=1}^{G} \delta(B'(|x_l| + |y_l|) + |w_l| + B'(|\frac{dL}{dx_l}| + |\frac{dL}{dy_l}|) + |\frac{dL}{dw_l}|) \\ &= 2\delta \sum_{l=1}^{G} \Big(\frac{B}{p}(|x_l| + |y_l|) + |w_l| \Big) \end{aligned} \tag{7}$$

2) Spatial parallelism: As mentioned in the previous section, the spatial dimensions of x, y, $\frac{dL}{dx}$ and $\frac{dL}{dy}$ are split among p PEs so that the memory at one PE is:

$$M_{spatial} = 2\delta \sum_{l=1}^{G} \Big(B\frac{(|x_l| + |y_l|)}{p} + |w_l| \Big) \tag{8}$$

Because each PE performs a computation with the size of the spatial dimensions as a fraction $\frac{1}{pw}$, $\frac{1}{ph}$, and $\frac{1}{pd}$ of the sequential implementation. This reduces the computation time of forward and backward phase of a layer by $p = pw \times ph \times pd$ times Thus, the computation time is:

$$\begin{aligned} T_{spatial,comp} &= \sum_{1}^{I} B \sum_{l=1}^{G} (\frac{FW_l}{p} + \frac{BW_l}{p}) + \sum_{1}^{I} \sum_{l=1}^{G} \Big(WU_l \Big) \\ &= \frac{D}{p} \sum_{l=1}^{G} (FW_l + BW_l) + \frac{D}{B} \sum_{l=1}^{G} (WU_l) \end{aligned} \tag{9}$$

The communication time includes the time to perform the Allreduce operation to share the weight gradients (similar to data parallelism) and the time to perform the halo exchange of each layer. For a layer l, a PE needs to send/receive the halo regions with the logically-neighboring PE(s). Thus the total time for halo exchange is

$$\begin{aligned} T_{spatial,halo} &= 2\frac{D}{B} \sum_{l=1}^{G} (T_{p2p}(B(\text{halo}(|x_l|))) + T_{p2p}(B(\text{halo}(|\frac{dL}{dy_l}|)))) \\ &= 2\frac{D}{B} \sum_{l=1}^{G} (2\alpha + B\delta\beta(\text{halo}(|x_l|) + \text{halo}(|\frac{dL}{dy_l}|))) \end{aligned} \tag{10}$$

In which halo() presents the size of data exchanged per batch. The exchanged data size depends on how each spatial dimension is split. We explain in detail our implementation for spatial parallelism in Section VII-B.

3) Layer parallelism: In this strategy, a DNN model is split into p composite layers (or group). Let g_i denote the group assigned to PE i. That is, each PE i keeps G_i layers of the model given that $\sum_{i=1}^{p} G_i = G$. Let FW_{G_i}, BW_{G_i}, and WU_{G_i} denote the time for performing the forward, backward, and weight update computation of group i, i.e., $FW_{G_i} = \sum_{l \in g_i} (FW_l)$, $BW_{G_i} = \sum_{l \in g_i} (BW_l)$, and $WU_{G_i} = \sum_{l \in g_i} (WU_l)$.

Pure implementation processes a batch of B samples at the first node and then sequentially pass the intermediate activation (gradients) through all p nodes in each iteration. Hence, the time for computation is:

$$\begin{aligned} T_{layer,comp} &= \sum_{1}^{I} (B \sum_{i=1}^{p} (FW_{G_i} + BW_{G_i}) + \sum_{i=1}^{p} (WU_{G_i})) \\ &= D \sum_{l=1}^{G} (FW_l + BW_l) + \frac{D}{B} \sum_{l=1}^{G} (WU_l) \end{aligned} \tag{11}$$

This approach does not reduce the computation time but it is helpful if the memory footprint at one node is limited. In practice, a pipeline implementation is used to reduce the computation time.

In a *pipeline implementation*, the mini-batch is divided into S segments of size $\frac{B}{S}$. In one stage, the computation of a layer group (or PE) g_i on a data segment s is performed simultaneously with the computation of layer group g_{i+1} on the data segment $s-1$, and so on. Thus, the time for each stage can be approximated by the maximum computation time of layer groups, i.e., $\max_{i=1}^{p}(FW_{G_i})$ or $\max_{i=1}^{p}(BW_{G_i})$. In

general, a pipeline implementation of p PEs with S data segments requires $(p+S-1)$ stages per iteration that leads to the total computation time of one epoch as:

$$T_{pipe,comp} \approx \frac{D(p+S-1)}{S}(\max_{i=1}^{p}(FW_{G_i}) + \max_{i=1}^{p}(BW_{G_i} + \max_{i=1}^{p}(WU_{G_i})) \tag{12}$$

Considering the communication in this strategy, each PE i has to pass forward/backward the output/input's gradients to the next/previous PE in a peer-to-peer communication scheme which costs $T_{p2p}(B|y_{G_i}|)$ and $T_{p2p}(B|\frac{dL}{dx_{G_i}}|)$, where y_{G_i} and x_{G_i} denote the output of the last layer and input of the first layer of a group layer g_i, respectively. In the pipeline fashion, the communication time of each stage can be approximated by $\max_{i=1}^{p-1} T_{p2p}(B|y_{G_i}|)$ and $\max_{i=2}^{p} T_{p2p}(B|\frac{dL}{dx_{G_i}}|)$. In the case of $|x_l| = |y_{l-1}|$, the total time for communication in one epoch ($I = \frac{D}{B}$ iterations) is summarized in:

$$T_{pipe,comm} \approx 2\frac{D(p+S-2)}{B}\Big(\max_{i=1}^{p-1}(\alpha + \frac{B}{S}|y_{G_i}|\delta\beta)\Big) \tag{13}$$

For the memory footprint, because each PE i stores a different set of layers, the maximum required memory in one PE is:

$$M_{pipe} = 2\delta \max_{i=1}^{p}\Big(2\sum_{l=1}^{G_i}(B(|x_l| + |y_l|) + |w_l|)\Big) \tag{14}$$

4) Filter parallelism: In this strategy, the computation time is reduced p times, yet the time for communication at a layer l becomes more complex, since it includes (1) an Allgather at the forward phase (except layer G)[3] that costs $T_{ag}(p, \frac{B|y_l|}{p})$, and (2) an Allreduce at the backward phase (except layer 1) that costs $T_{ar}(p, B|\frac{dL}{dx_l}|) = T_{ar}(p, B|x_l|)$. In the case of $|x_l| = |y_{l-1}|$, the total time for communication in $I = \frac{D}{B}$ iterations is:

$$T_{filter,comm} = 3\frac{D}{B}(p-1)\sum_{l=1}^{G-1}(\alpha + \frac{B|y_l|}{p}\delta\beta) \tag{15}$$

In this strategy, each PE keeps only $\frac{1}{p}$ the filters (weight) of each layer. However, PE i needs to communicate with other PEs to share its local partial activations, hence requiring memory to store the entire activation $|y_k|$. The required memory at each PE is:

$$M_{filter} = 2\delta\sum_{l=1}^{G}\Big(B(|x_l| + |y_l|) + \frac{|w_l|}{p}\Big) \tag{16}$$

5) Channel parallelism: Similar to the filter parallelism, channel parallelism splits the DL models horizontally, i.e., by the number of input channels C. Thus, the computation time, and the required memory at each PE are same as those of filter parallelism

$$M_{channel} = 2\delta\sum_{l=1}^{G}\Big(B(|x_l| + |y_l|) + \frac{|w_l|}{p}\Big) \tag{17}$$

$$T_{channel,comp} = T_{filter,comp} = \frac{D}{p}\sum_{l=1}^{G}\Big(FW_l + BW_l\Big) + \frac{D}{pB}\sum_{l=1}^{G}(WU_l) \tag{18}$$

The communication is performed in a different pattern that includes (1) an Allreduce at the forward phase (except layer G) that costs $T_{ar}(p, B|y_l|)$, and (2) an Allgather at the backward phase (except layer 1) that costs $T_{ag}(p, \frac{B|\frac{dL}{dx_l}|}{p})$. Similar to filter parallelism, we get the total communication time:

$$T_{channel,comm} = 3\frac{D}{B}(p-1)\sum_{l=1}^{G-1}(\alpha + \frac{B|y_l|}{p}\delta\beta) \tag{19}$$

6) Hybrid parallelism (Data + Filter): We consider an example of hybrid parallelism: the combination of data and filter parallelism in which we use $p1$ data parallelism groups in $p = p1 \times p2$ PEs. We apply filter parallelism inside each group and data parallelism between groups. Each group will process a partition of the dataset, i.e., $\frac{D}{p1}$ samples. Each PE then keeps one part of filters of each layer, e.g., $\frac{F}{p2}$ filters, so that the required memory is:

$$M_{df} = 2\delta\sum_{l=1}^{G}\Big(\frac{B}{p1}(|x_l| + |y_l|) + \frac{|w_l|}{p2}\Big) \tag{20}$$

Each PE hence performs $\frac{1}{p2}$ of the computation at each layer with a mini-batch of $\frac{B}{p1}$. The computation time is:

$$\begin{aligned} T_{df,comp} &= \sum_{1}^{I}\frac{B}{p1}\sum_{l=1}^{G}\Big(\frac{FW_l}{p2} + \frac{BW_l}{p2}\Big) + \sum_{1}^{I}\sum_{l=1}^{G}(\frac{WU_l}{p2}) \\ &= \frac{D}{p}\sum_{l=1}^{G}\Big(FW_l + BW_l\Big) + \frac{D}{Bp2}\sum_{l=1}^{G}(WU_l) \end{aligned} \tag{21}$$

In this strategy, the communication includes intra-group and inter-group communication, which correspond to the cases of filter and data parallelism. The total communication time of one iteration includes $T_{ag}(p_2, \frac{B|y_l|}{p2})$ and $T_{ar}(p_2, B|\frac{dL}{dx_l}|)$ at each layer and $T_{ar}(p_1, \sum_{l=1}^{G}\frac{|w_l|}{p2}|)$ when update, respectively. The total communication time becomes:

$$\begin{aligned} T_{hybrid,comm} = 3\frac{D}{B}(p2-1)\sum_{l=1}^{G-1}(\alpha + \frac{B|y_l|}{p}\beta) + \\ 2\frac{D}{B}(p1-1)(\alpha + \frac{\sum_{l=1}^{G}|w_l|}{p}\beta) \end{aligned} \tag{22}$$

We summarize our analytical model in Table III.

TABLE VI: Implementation Overview(✓: customized; - : untouched)

Parallelism strategy	Conv	Pool	BNorm LNorm	ReLU	FC	GEMM
Data	-	-	-	-	-	-
Spatial	✓	✓	-	-	✓	-
Filter / Channel	✓	-	-	-	✓	✓

B. Empirical Experiments Detail

1) Implementation Details: We implement data/channel/filter and spatial parallelism strategies using Chainer v7.0.0 [52] with CUDA v10.0 and ChainerMN [53] for distributed execution. Although Chainer provides a built-in implementation for data parallelism and some level of support for model parallelism, it is not sufficient for testing all the parallel strategies we study here (the same insufficiency also goes for PyTorch and TensorFlow). Significant engineering effort was required to modify the existing implementation to support all forms of parallelism. We mark our implementation for each type of targeted layers and parallel strategies in Table VI.

More specifically, we use the default implementation of Chainer for data parallelism. Since the size of each dimension (i.e., H, W, and/or D) limits the parallelism of spatial strategy, in this work, we implement the spatial strategy for some first layers of a given model until adequate parallelism is exposed while still maintaining the maximum required memory per

[3]Each process i transfers $|(y_l)_i[*, p, *]| = \frac{|y_l|}{p}$ values for one sample in layer l, and a total of $B\frac{|y_l|}{p}$ values for the entire batch.

node within memory capacity. We then implement an Allgather to collect the full set of activations before passing it to the following layers which perform similar to the sequential implementation. For example, we aggregate after the final convolution layer (before a fully-connected layer) in VGG16, and ResNet50. For CosmoFlow, we aggregate after the second convolution/pooling layer because most of required memory footprint and compute are in those first two layers.

The minimum number of input channels C at each layer limits the parallelism of channel strategy, e.g., only 3 input channels in the case of ImageNet dataset. In this work we implement the channel parallelism from the second layers.

For hybrid strategies such as data plus filter (or data plus spatial), we map the data parallel part on inter-node. This implementation is also used in [34] which implements filter parallelism inside one node and data parallelism between nodes. We leverage ChainerMN with MPI support for both inter-node and intra-node communication. To perform an Allreduce and update the gradients in data and spatial parallelism, we use ChainerMN's multi-node optimizer which wraps an optimizer and performs an Allreduce before updating the gradients. For the hybrid, we perform a reduce inside each node to a leader GPU first then perform a global Allreduce between the leaders. These two Allreduce are different as they involve different parallelization techniques (e.g., spatial in local and data in global). Note that we implement all communication functions based on Nvidia NCCL library [40] (version 2.4.8.1), except for the Allgather and halo exchange of spatial strategies. It is because of the NCCL library does not support peer to peer and Allgatherv communication interface.

2) Accuracy and Correctness: We aim at making sure our implementation of different parallelism strategies have the same behavior as data parallelism. We first compare the output activations/gradients (in forward/backward phases) of each layer (value-by-value) to confirm that the parallelization artifacts, e.g., halo exchange, do not affect the correctness. It is important to note that the implementations for different parallel strategies change only the decomposition of the tensors, and do not change any operator or hyperparamaters that have an impact on accuracy (as elaborated in Section 4.1 in main manuscript).

Second, in this work we assure that batch normalization (BN) layers are supported in all parallel strategies since the accuracy of training can be affected by the implementation of the BN layers [54], [22]. More specifically, for the data parallelism strategy, the typical implementations of batch normalization in commonly used frameworks such as Caffe, PyTorch, TensorFlow are all un-synchronized. This implementation leads to data being only normalized within each PE, separately. In normal cases, the local batch-size is usually already large enough for BN layers to function as intended. Yet in some cases, the local batch-size will be only 2 or 4 in each PE, which will lead to significant sample bias, and further degrade the accuracy. In this case, we suggest to use the synchronized BN implementation as mentioned in [22],

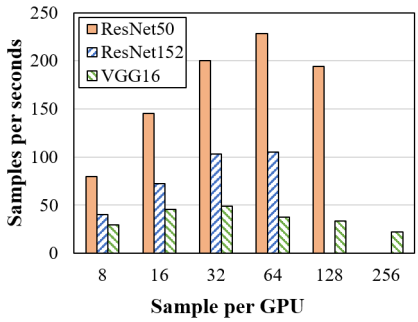

Fig. 9: Performance vs. sample per GPUs (data parallelism).

which requires a communication overhead for computing the global mini-batch mean.

In spatial strategy, although performing batch normalization on subsets of the spatial dimensions has not been explored, to the authors knowledge, this computation requires no significant adjustment [37]. More specifically, BN is typically computed locally on each PE on its own portion of the spatially partitioned data.

In filter and channel parallelism strategy, since all PEs keep the same set of activations after performing the Allgather operation at each layer, the BN layer could be implemented as in the sequential strategy. It could be implemented in a centralized fashion, (e.g., one PE performs the BN and then send the result to other PEs). Alternatively, each node could redundantly compute the BN layer (distributed approach). In this work, we assume the use of the distributed approach which does not require any communication overhead, yet replicates computation.

3) Configurations of Experiments: An important performance factor is efficient device occupancy of GPUs. Thus, we conducted a series of test runs for each type of parallelism and DL model to identify the optimal number of samples per GPU (or node) that would efficiently utilize the device. Figure 9 show the training performance versus the number of samples per GPUs in data parallelism. We observed that the performance drops significantly when we train using a higher samples/GPU number than the optimal one. This occurs when the computational load becomes too large to effectively utilize a single GPU.

In addition, in CosmoFlow with spatial strategies, although we use only one sample per node, i.e., 0.25 samples/GPU, we could not avoid the performance drop. We conducted a small experiment to examine the effect of non-optimal number of sample on the accuracy of ParaDL. Table VII shows significant performance improvement when using a synthetic 256^3 size, in comparison to the the performance when using the original sample size (512^3). This experiment indicates that future in-depth investigation and alternatives are required for large 3D samples that are beyond the optimal parallelization size, e.g. strong scaling for specific ranges of sizes, or split spatially across nodes in a hierarchy to reduce halo exchange overheads. One could reduce the problem size per GPU by dividing sample among other GPUs outside the node, yet this can

TABLE VII: Average time in second and performance (block/second) for training CosmoFlow with a block of one sample of size and 512^3 and a block of 8 synthetic samples of size 256^3. One sample is process per iteration.

#Samples	Size	16 GPUs		32 GPUs	
		Time	Performance	Time	Performance
1	512^3	6.192	0.16	6.973	0.14
8	256^3	0.177×8	0.71 (4.4×)	0.212×8	0.59 (4.2×)

cause high inter-node communication overhead, especially, for performing halo exchange and Allgather. In this work, we choose to implement the approach of fixing 4 GPU for spatial splitting, since it fits well with the hybrid parallelism strategy.

C. Analytical model's Projection and Accuracy

In this section we first give an assessment of the capability of the analytical model as prediction utility, of both performance and scaling, by evaluating the accuracy and robustness of the oracle. For that we run a wide range of experiments with different configurations of models, resources, and parallel strategies.

1) Projection accuracy: This section discusses how close is the projection of the analytical model in comparison with the measured experiments. It is a complex task to accurately project the performance of DL training, especially when scaling. More specifically, the following factors have a significant effect on performance: contention on the PFS, the effectiveness of the pipeline used for asynchronous data loading, network contention, implementation quality, and overheads of solution fidelity book-keeping. That being said, in this section we aim to demonstrate that the presented oracle, despite the complexities mentioned above, reasonably represents the reality of measured runs on an actual system (especially when scaling to a 1024 GPUs). In this comparison, we focus only on the computation and communication time of the main training loop excluding other times from this comparison such as I/O staging or others.

Figure 2 (in the main text) shows the oracle's projections versus the measured runs for different parallel strategies, and using four different models. We ran all the permutations of possible configurations but plot only some of them because of space limitations. The iteration time is calculated as an average of 100 iterations excluding the first iteration which normally involves initialization tasks. We try to avoid the issue of congestion as much as possible by running several runs for each data points and choosing the one with stable values. We define the relative ratio of the projection result of our analytical model (ParaDL) to the empirical result as the (*projection accuracy*). We scale the tests from 16 to 1024 GPUs for data and hybrid parallelism and from 4 to 64 GPUs for filter/channel parallelism. To get a more detailed analysis, we decompose the execution time into computation and communication. Overall the accuracy of ParaDL predictions is reasonably high, given the complexity in projecting for a production system with 1000s of GPUs, with an average of 0.84x for Data-ResNet50, 0.76x for Data-VGG16, 0.75x for Channel-ResNet50, 0.69x for Filter+Data-ResNet50, and 0.60x for Spatial+Data-ResNet50.

As we can see, the large majority of measured computation times are very well aligned with ParaDL predictions. For the communication, the empirical fits well for a lower number of GPUs but then slightly diverges from the prediction as we scale to more than 512 GPUs. This network congestion at large scale occurs when communicating via the 1:3 oversubscription network topology and pushes the communication time up away from the oracle predictions (See Section V.B.1-Network congestion in the main text).

For the computation, it fits well for data parallelism because data parallelism splits the dataset and does not change the computation logic. For model parallelism, such as channel/filter or hybrid of filter+data, the projection time of ParaDL shows the ideal cases when the computation time is reduced linearly, to the number of GPUs. However, our implementation could not achieve such ideal scaling due to lack of support from the underlying framework, which is only optimized for data parallelism. For example, the computation of a convolution layer barely changes with the increase in number of filters as shown in Figure 6 (main text).

Another remarkable tradeoff for model and hybrid parallelism is the efficient utilization of GPUs. As mentioned in Section VII-B, there exists an issue in using a non-optimal number of samples per GPU with CosmoFlow hence it makes the measured run communication significantly high. In that case, the accuracy ParaDL becomes worse, i.e., approximately 0.3x as in Figure 2 (main text). We conducted a small experiment to examine the effect of non-optimal number of sample on the accuracy of ParaDL. Table IX shows significant accuracy improvement when using a synthetic 256^3 size, in comparison to the the accuracy when using the original sample size (512^3).

2) Ranking of parallel strategies: This section examines the ranking accuracy of the ParaDL, i.e., how accurate is ParaDL when suggesting a ranking for different parallel strategies at a fixed GPU budget and when scaling. Table VIII shows the ranking proposed by the oracle versus that of measured runs with different parallel strategies for ResNet50 and VGG, from 32 up to 256 GPUs, respectively. The table also shows the computation, communication, and total time in seconds for one epoch, for each configuration. Overall, although there's some gap between ParaDL and empirical results (such as computation time of Filter), the ranking accuracy of ParaDL is correct at 100%. The discrepancy is in the channel vs. filter parallelism; two very similar strategies. ParaDL ranks those parallelism strategies the same rank but filter is faster than channel in our specific implementation.

When scaling, there exists a limit on the number of GPUs p for each of the model parallel strategies (last column of Table III). For example, p can not exceed the minimum number of filters of a layer in the model, i.e., 64 in the case of VGG and ResNet50 with the filter parallelism strategy. Therefore, we consider only hybrid strategies with 128 and 256 GPUs. For VGG, we use the same number of samples per node for both of the hybrid strategies. The performance of filter+data exceeds that of spatial+data due to less of intra-node

TABLE VIII: Measured and projected rankings by performance for training ResNet50 and VGG with $32-256$ GPUs.

Strategies	Empirical				ParaDL				Empirical				ParaDL			
	Comp	Comm	Total	Rank	Comp	Comm	Total	Rank	Comp	Comm	Total	Rank	Comp	Comm	Total	Rank
	ResNet50 - 32 GPUs								**VGG - 32 GPUs**							
Spatial + data	65	416	481	1	130	205	335	1	1039	10354	11393	2	328	5265	5593	2
Filter + data	457	306	763	2	144	528	672	2	760	1547	2307	1	730	1770	2500	1
Filter	2390	7580	9970	3	321	13591	13912	3	9033	77266	86299	3	694	18542	19236	3
Channel	2988	14579	17567	4	321	13591	13912	3	49109	59704	108814	4	694	18542	19236	3
	ResNet50 - 64 GPUs								**VGG - 64 GPUs**							
Spatial + data	33	211	244	1	65	95	150	1	513	5178	5691	2	164	2882	3046	2
Filter + data	232	163	395	2	72	159	231	2	376	779	1156	1	365	922	1287	1
Filter	6676	14988	18364	3	262	18469	18731	3	32102	70341	102443	3	13275	20822	34097	3
Channel	3273	17574	20858	4	262	18469	18731	3	49929	60284	110213	4	13275	20822	34097	3
	ResNet50 - 128 GPUs								**VGG - 128 GPUs**							
Spatial + data	17	119	136	1	32	53	85	1	232	2718	2950	2	82	1421	1503	2
Filter + data	115	119	233	2	36	140	175	2	187	421	607	1	182	473	655	1
	ResNet50 - 256 GPUs								**VGG - 256 GPUs**							
Spatial + data	9	64	73	1	16	27	43	1	117	1416	1533	2	41	720	761	2
Filter + data	60	64	124	2	18	72	90	2	96	217	313	1	91	241	332	1

TABLE IX: Average iteration time (second) for training CosmoFlow with one synthetic samples of size 256^3 and 512^3.

Size	16 GPUs			32 GPUs		
	Emp.	Para.	Acc.	Emp.	Para.	Acc.
256^3	0.177	0.162	0.91	0.212	0.164	0.77
512^3	6.192	2.176	0.33	6.973	2.178	0.31

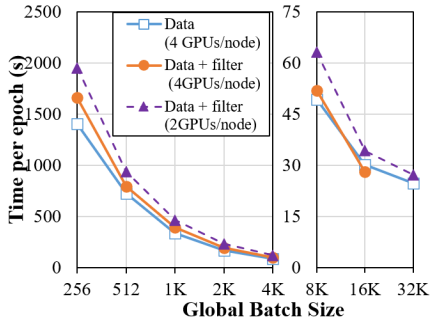

Fig. 10: Hybrid filter + data scaling with ResNet50 on the system of 4 and 2 GPUs per node. Data is not always faster in large scale.

communication. However, for ResNet50, these two rankings are swapped. Although the time per iteration of filter+data is still less than that of spatial+data, the latter hybrid strategy can run with a higher number of samples per node, e.g., 256 vs. 64, yet it reduces the number of iterations and the overall epochs time.

D. Tradeoffs for Performance and Scalability

In this section we highlight tradeoffs and considerations of performance and scalability that we observed when experimenting with different model parallel strategies. **Spatial+Data Hybrid for 3D Convolutions**: 3D convolutions (e.g. CosmoFlow), are typically associated with large sample sizes (very relevant to the datasets in computational sciences). The typical approach of parallelism used in 3D convolution is spatial parallelism [55]; data parallelism is not feasible when data samples are too large. However, the limiting factor for spatial is that the number of GPUs could exceed the parallelized dimension(s). Splitting more dimensions (e.g. 3 dimensions rather 2) comes at an extra cost of increasing the number of neighbors engaged in halo exchange and higher risk of errors. As Figure 7 shows, using spatial+data hybrid is an effective scalable alternative (despite communication inefficiencies), since it scales both in performance and GPU count (i.e. one could simply expand the data parallel pool as much as new nodes are added). Indeed, the curve shows a perfect scaling (Note the logarithmic y-axis). It is worth mentioning that this is the first multi-GPU implementation of ComsoFlow, since the original implementation used Intel KNL as standalone CPUs. As more datasets from HPC start to be analyzed by DL frameworks, this type of hybrid parallelization strategies will become more and more relevant because data parallelism will be simply not enough.

Data Parallelism Not Always Faster: Data parallelism is reported to outperform other model and hybrid parallel methods [33], for the same number of nodes. Interestingly, our results show that there can be cases where filter+data hybrid can outperform data parallelism at large scale. Figure 10 shows the scaling of data vs other hybrid modes while scaling ResNet50. As we can see, at a global batch size of 16K, data+filter hybrid is 8% faster than data parallelism. This is due to the advantage of the segmented collectives inside the node over the data parallel Allreduce that scales unfavorably.

Density of GPUs in a Node: We also compare the runtime of two configurations of data+filter hybrid: same batch size but different number of GPU for the same number of nodes). We observed that 4 GPUs is only 20% faster than 2 GPUs. This implies that less dense systems can be more cost effective for this type of hybrid parallelism.