An Oracle for Guiding Large-Scale Model/Hybrid Parallel Training of Convolutional Neural Networks

Albert Njoroge Kahira*
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
Barcelona, Spain
albert.kahira@bsc.es

Truong Thao Nguyen*
National Institute of Advanced Industrial Science and Technology
Japan
nguyen.truong@aist.go.jp

Leonardo Bautista Gomez
Barcelona Supercomputing Center
Barcelona, Spain
leonardo.bautista@bsc.es

Ryousei Takano
National Institute of Advanced Industrial Science and Technology
Japan
takano-ryousei@aist.go.jp

Rosa M Badia
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
Barcelona, Spain
rosa.m.badia@bsc.es

Mohamed Wahib
National Institute of Advanced Industrial Science and Technology
RIKEN-CCS
Japan
mohamed.attia@aist.go.jp

ABSTRACT

Deep Neural Network (DNN) frameworks use distributed training to enable faster time to convergence and alleviate memory capacity limitations when training large models and/or using high dimension inputs. 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 of Convolutional Neural Networks (CNNs) to understand the trade-offs between different parallelism approaches on performance and scalability. We leverage our model-driven analysis to be the basis for an oracle utility which can help in detecting the limitations and bottlenecks of different parallelism approaches at scale. We evaluate the oracle on six parallelization strategies, with four CNN models and multiple datasets (2D and 3D), on up to 1024 GPUs. The results demonstrate that the oracle has an average accuracy of about 86.74% when compared to empirical results, and as high as 97.57% for data parallelism.

CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**; **Distributed computing methodologies**; *Machine learning*.

KEYWORDS

Deep Learning, Model Parallelism, Performance Modeling

*Both authors contributed equally to this paper.

The final publication is availabl at ACM via:
https://doi.org/10.1145/3431379.3460644

ACM Reference Format:
Albert Njoroge Kahira, Truong Thao Nguyen, Leonardo Bautista Gomez, Ryousei Takano, Rosa M Badia, and Mohamed Wahib. 2021. An Oracle for Guiding Large-Scale Model/Hybrid Parallel Training of Convolutional Neural Networks. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '21), June 21–25, 2021, Virtual Event, Sweden.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3431

1 INTRODUCTION

DNNs are achieving outstanding results in a wide range of applications, including image recognition, video analysis, natural language processing [45], understanding climate [21], and drug discovery [50], among many others. In the quest to increase solution accuracy, researchers are increasingly using larger training datasets as well as larger and deeper DNN models [3, 17, 54]. In addition, applying Deep Learning (DL) in new domains, such as health care and scientific simulations, introduce larger data samples and more complex DNN models [26]. 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 times for larger models and datasets [3]. 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 [9], those efforts were premature and remained far from production deployments, since data parallelism was simple and sufficient. However, the growth in datasets and models far outgrows the increase in compute capability [17]. Accordingly, scaling data parallelism can be limited by the memory capacity and the communication overhead. *First,* we elaborate on the memory capacity issue. In data parallelism, the entire model is duplicated for each compute node. Therefore, training larger and deeper neural networks have to deal with the memory capacity limits. A notable case is in the area of language modeling at which models are increasingly approaching $O(100B)$ parameters [38] (ex: GPT-3 has 175B parameters [4]). In addition, for sample sizes with higher dimensions, e.g., 3D scientific data sets [33] and videos, the memory capacity would also

limit the number of samples that can be concurrently processed by a GPU [31] hence, restricting the scaling of data parallelism. *Second,* we elaborate on the communication overhead. A major bottleneck when scaling data parallelism is the large-message Allreduce collective communication for the gradient exchange at the end of each iteration [18, 53]. Several active efforts try to optimize the Allreduction collective algorithm for supporting large messages on specific network architectures in HPC systems [2, 34, 53]. However, even with those algorithms, communication remains a bottleneck when the size of the models increases. Moving to model or hybrid parallelism is one of the ways to reduce this communication overhead [14].

Given those issues with data parallelism and the growing scale of training, researchers are tackling different bottlenecks across the different components necessary for distributed DNN training. Table 1 shows a summary of the recent approaches in scaling distributed DNN training, split into components and training phases. **Despite those efforts, data parallelism is not feasible for all cases.** Thus, it is important to understand the limitations and scalability of large scale model and hybrid parallelism training of DNNs.

In this work, we focus on the HPC aspects of scaling six different strategies for model and hybrid parallelism in CNNs distributed training. Innovations in DL theory (e.g., optimizers) are out of the scope of this paper. While most works in the literature focus on improving the performance of one single parallelism strategy for one specific framework; our study functions as the basis for a tool, named *ParaDL*, capable of modeling and predicting the performance of a large set of configurations for CNN distributed training at scale. In addition, ParaDL also helps to reveal the practical limits and bottlenecks of different parallel strategies in CNN training .

Our main contributions in this work are as follow:

- We formally define the main parallel strategies (See Section 3), including hybrid ones, and provide a comprehensive analysis of the compute, communication, and memory footprint when training CNNs for inputs of any dimension.
- We propose an oracle (ParaDL) [1] that projects the ideal performance of distributed training of CNNs, broken down by training phases. This helps in favoring a parallel strategy on a given system (See Section 4) and aids in identifying optimization opportunities in frameworks.
- We implement all parallelization strategies to validate our model, except for: a) data parallelism (already supported by most DL frameworks), and b) using an existing pipeline implementation.
- We show the utility of ParaDL in exposing performance and scalability trade-offs. The accuracy of ParaDL (86.74% on average and up to 97.57%) is demonstrated by conducting a wide range of experiments for different CNN models, parallel strategies, and datasets, on up to 1,024 GPUs (See Section 5).

2 BACKGROUND

2.1 Phases of Distributed Training of DNNs

2.1.1 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 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.

2.1.2 Components of Distributed Training. To optimize for the performance and efficiency of training at 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**-systems.

2.2 Notation

We summarize our notation in Table 2. In a G-layer CNN 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 bias $bi_l[F_l]$.
- 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 in [14]). 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 $w[C, F, 0]$.

[1] https://github.com/TruongThaoNguyen/paraDL-analysis

Table 1: 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 [37] (fewer epochs to converge, but longer iterations and more memory).
Normalization	●	○	○	○	○	-	✓	-	-	Cross-GPU Batch-normalization [55] (requires extra communication) and Group Normalization [51].
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 [2, 34, 53] (data parallelism)
Sparsification	○	●	●	○	○	-	✓	✓	✓	Reducing the computation volume [29] or/and communication message size [39] 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) [43], gradient checking point [6], out-of-core methods [49],
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 [10] or BiGraph [12].
Model/hybrid parallelism (our target in this work)	○	○	●	○	○	✓	✓	✓	✓	Castello et al. [5] analyzed the communication trade-offs in some model parallel strategies. Hybrid parallelism are proposed in [15] (spatial with data) and [14] (channel/filter with data). [19, 56] explored different parallelization schemes on per-layer basis.

Table 2: Parameters and Notation

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
bi_l	Biases 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 conv. 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
γ	Memory reuse factor

The sequential implementation of CNN 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) $\frac{dL}{dw}[C, F, K] \leftarrow BW_{weight}(\frac{dL}{dy}[N, F, Y], x[N, C, X])$

At the end of each iteration, weights of all layers are updated:

(WU) $w[C, F, K] \leftarrow WU(\frac{dL}{dw}[C, F, K], \rho)$

3 STRATEGIES FOR DISTRIBUTED TRAINING

Training CNNs using a single processing element (PE) is computationally expensive, e.g., training ResNet-50 over a single V100 GPU requires 29 hours. Hence distributed training on HPC systems is common for large models and datasets. Parallelizing of training process should be done by splitting different dimensions. 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 CNNs: (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*) [13], (3) vertically partitioning the neural network along its depth (*layer parallelism*) and overlapping computation between one layer and the next layer [17] (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*) [13, 15]. In addition, a combination of two (or more) types of the mentioned parallelism strategies is named as *hybrid parallelism* (e.g., Data+Filter parallelism and Data+Spatial parallelism, or *df* and *ds* respectively for short, are some examples of hybrid parallelism).

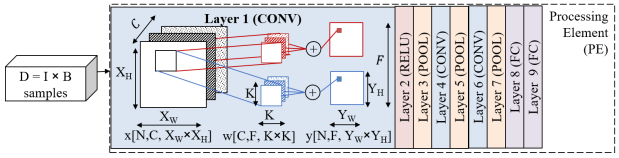
In this work, when presenting tensors such as x, y, and w, we use the $*$ symbol to present a dimension for which its values are replicated between processes. To emphasize that a tensor's dimension is partitioned among different PEs, we use the number of processes p. For example, in data parallelism, $x[p, *, *]$ 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 Allreduce communications.

It is important to note that the notation and analysis in this paper are general to input tensors of any dimension (1D, 2D, and, 3D). Input tensors of higher dimensions are also valid in our analysis since they can be represented as 3D tensor with the extra dimensions as component vector(s) (e.g. CosmoFlow [31] 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.

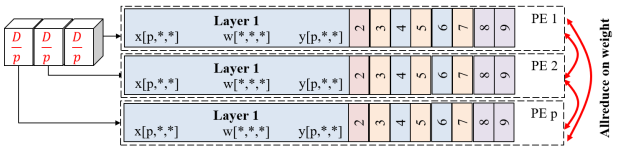
3.1 Data parallelism

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:

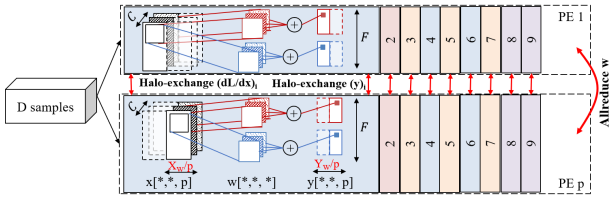
(IO) $(x)_i[p, *, *] \leftarrow IO(\text{sub-dataset}_i, B')$ in the first layer.
(FB) $(y)_i[p, *, *] \leftarrow FW(x_i[p, *, *], w[*, *, *])$
(FB) $(\frac{dL}{dx})_i[p, *, *] \leftarrow BW_{data}((\frac{dL}{dy})_i[p, *, *], w[*, *, *])$
(FB) $(\frac{dL}{dw})_i[*, *, *] \leftarrow BW_{weight}((\frac{dL}{dy})_i[p, *, *], (x)_i[p, *, *])$



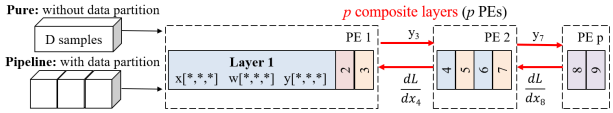
(a) Sequential implementation on a single PE



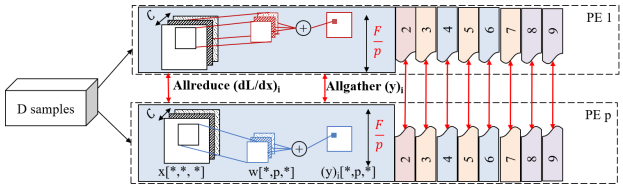
(b) Data parallelism



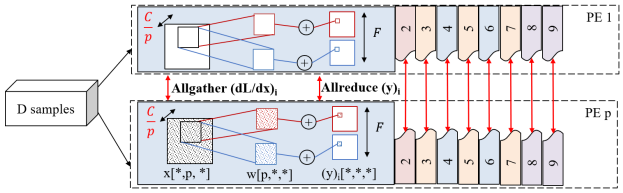
(c) Spatial parallelism splits the input x and output y on either width (as shown in this figure), height or both dimensions



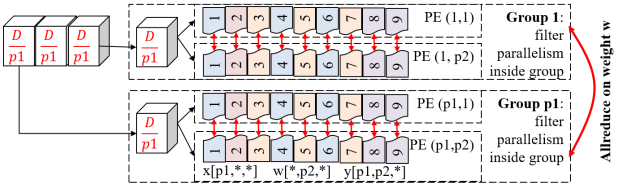
(d) Layer parallelism (partition the model vertically with pipeline implementation)



(e) Filter parallelism (partition the model horizontally)



(f) Channel parallelism (partition the model horizontally)



(g) Hybrid parallelism (example of filter on top of data parallelism)

Figure 1: Different strategies for distributed training of CNNs. Red solid lines refer to communication

(GE) $\frac{dL}{dw}[*,*,*] \xleftarrow{Allreduce} \sum_{i=1}^{p}((\frac{dL}{dw})_i[*,*,*])$
(WU) $w[*,*,*] \leftarrow WU(\frac{dL}{dw}[*,*,*])$

3.2 Spatial parallelism (height-width-depth)

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) [13]. The exchanged data size (i.e., halo(x_l)) depends on how each spatial dimension is split, and the stride length. 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 for 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) $\frac{dL}{dw}[*,*,*] \xleftarrow{Allreduce} \sum_{i=1}^{p}((\frac{dL}{dw})_i[*,*,*])$
(WU) $w[*,*,*] \leftarrow WU(\frac{dL}{dw}[*,*,*])$

3.3 Model-horizontal parallelism

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 [15] 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[2]). Because each PE performs the weight-update on its portion of weights, the gradient-exchange phase is skipped.

[2]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 [14].

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

3.4 Model-vertical (layer) parallelism

A model parallel variant at which the CNN 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 [17]). 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.

3.5 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) $y_j[p1,*,*] \xleftarrow{Allgather} \bigcup_{i=1}^{p2}((y)_{ij}[p1,p2,*])$
(FB) $(\frac{dL}{dx})_{ij}[p1,*,*] \leftarrow BW_{data}((\frac{dL}{dy})_{ij}[p1,p2,*], w[*,p2,*])$
(FB) $(\frac{dL}{dx})_j[p1,*,*] \xleftarrow{Allreduce} \sum_{i=1}^{p2}((\frac{dL}{dx})_{ij}[p1,*,*]))$
Data parallelism between $p1$ groups :
(FB) $(\frac{dL}{dw})_j[*,p2,*] \leftarrow BW_{weight}((\frac{dL}{dy})_{ij}[p1,p2,*], (x)_{ij}[p1,*,*])$
(GE) $\frac{dL}{dw}[*,p2,*] \xleftarrow{Allreduce} \sum_{j=1}^{p1}((\frac{dL}{dw})_j[*,p2,*])$
(WU) $w[*,p2,*] \leftarrow WU(\frac{dL}{dw}[*,p2,*])$

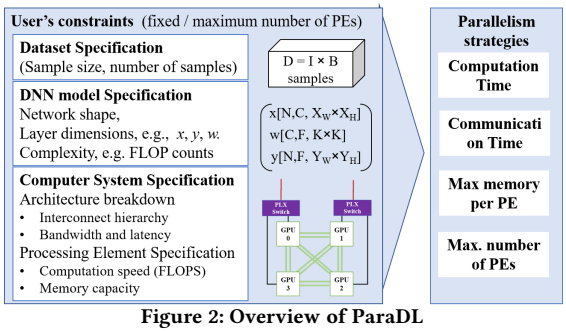


Figure 2: Overview of ParaDL

Another example of hybrid parallelism is the combination of data and spatial or channel parallelism [14]. Furthermore, the hybrid strategy could be more complex when applying different parallel strategies for different layers [19, 25].

4 PERFORMANCE PROJECTION OF DIFFERENT PARALLEL STRATEGIES

4.1 Overview of ParaDL

In this section we introduce our oracle (*ParaDL*). Through the information that we can get beforehand, such as the dataset, model, supercomputer/cluster system specification, and user's constraints (e.g., maximum number of involved PEs), ParaDL calculates the computation and communication time to project the overall performance (as described in Figure 2). If the strategy differs as the number of nodes increases, ParaDL would breakdown the execution time of different strategies as the number of PEs changes, i.e. scaling the number of PEs. ParaDL can be used for the following purposes:

- Suggesting the best strategy for a given CNN, dataset, and resource budget (especially when data parallelism is not feasible).
- Identifying the time and resources to provision from a system (we partially relied on ParaDL in this paper for that purpose when conducting our empirical experiments in Section 5).
- Comparison of projections with measured results to detect abnormal behavior (we relied on ParaDL for this purpose in our analysis of network contention in Section 4.3).
- Identifying limitations of parallel strategies, shortcomings of frameworks, and bottlenecks in systems (we relied on ParaDL for this purpose in our discussion in Section 5.3).
- As an education tool of the parallel strategies that would improve the understanding of parallelism in DL

Frameworks that are used for DL are comprised of complex and interleaved layers of optimized functions. A pure analytical model of parallel strategies in CNNs would, therefore, be impractical. In this paper we adopt a hybrid analytical/empirical modeling approach at which we: (i) use analytical modeling for functional requirements driven by the parallelism strategies (Section 4.3), and (ii) empirical parametrization for functions not related to the parallel strategy being deployed (more details in Section 4.4). Finally, we quantify the accuracy of the oracle with a large empirical evaluation in Section 5.

4.2 Assumptions and Restrictions

The study in this paper is based on the following assumptions.

Targeted models and datasets: our study covers all types of layers used in production CNNs, and could hence be used for projecting the performance of any production CNN model, not just the models we evaluate in the paper. We also support the input (i.e. samples) to be of any dimension (as shown in Table 2).

Training time and memory estimation: Our study focuses predominantly on the computation and communication time of the CNN training, thus we assume that all the training data is available in memory before starting the training process. In other words, in this model we do not include the time for I/O.

One could conservatively estimate the memory required on a per layer basis by assuming the memory buffers of the output of layer l are different from the memory buffers for the input of layer $l+1$, however, in reality both buffers being the same. Additionally, in reality there is a variety of optimizations that frameworks implement to reduce the memory used (See Table 1). Since those optimization methods are complexly intertwined and depend on the framework implementations, without loss of generalization, we propose a practical memory requirement estimation. More specifically, we start out from the naive memory projection that aggregates layers, then we reduce that conservative upper bound to reflect the actual memory optimizations happening inside frameworks. We introduce a memory reuse factor γ. The actual minimum required memory, after all memory reuse optimizations are applied, can be estimated by multiplying total naive required memory by γ. This memory reuse factor can be derived from several elaborate studies on model-level and layer-level memory profiling of CNNs [20, 28].

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 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 [25].

4.3 Performance and Memory Projection

In this section, we estimate the total training time in one epoch and maximum memory per PE for the mentioned main parallel strategies, including hybrid. 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 [3]. $T_{ar}(p,m)$, $T_{ag}(p,m)$, and $T_{p2p}(m)$ stand for the time of transferring a data buffer of m-size between p PEs via an Allreduce, Allgather, and a peer-to-peer scheme, respectively. In data parallelism, the training 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 needs memory to store its input $B'|x_l|$, activation $B'|y_l|$, weights $|w_l|$, bias $|bi_l|$, the gradients $B'|\frac{dL}{dx_l}|$, $B'|\frac{dL}{dy_l}|$, and $|\frac{dL}{dw_l}|$. Overall, if each item of the input, activation, weight and gradients are stored in δ bytes, the maximum required memory at one PE is:

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

In theory, the computation time can be estimated by observing the dataset and CNN model (e.g., FLOP counts and the computation speed of each PE). For modeling the communication time, there exist various derived / specific analytical performance models, e.g., as in the survey of Rico Gallego et. al. [40]. To keep the performance modeling generic, we choose to use the Hockney $\alpha - \beta$ model, in which, the peer-to-peer communication time of transferring a message of size m is modeled by $T_{p2p}(p,m) = \alpha + m\beta$. Time for a message send from a source to a destination is α (also known as startup time) and the time to inject one byte of data into the network is β. We follow the common practice in DL communication libraries such as NCCL [35] to use a ring-based algorithm for all the collective communication operation with large message sizes and a tree-based algorithm for small message sizes. In the ring-based 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 the maximum memory required per PE for the mentioned parallel strategies. We summarize our analytical model in Table 3 [4]. The details of this analysis can be found in the Appendix of this paper.

Contention modeling: Ideally, in a system without contention, the start up time α of a given pair is estimated as the total switching latency, which 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). However, network congestion is one of the biggest problems facing HPC systems today, affecting system throughput and performance. To address the contention effects we introduce the use of a contention penalty coefficient ϕ, which divides the bandwidth of a link by the number of communication flows ϕ sharing this

[3] 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.

[4] We note that the proposed communication model can naturally be extended for different Allreduce schemes or algorithms, such as Parameter Server or tree-based algorithms. For example, when message sizes are small, communication time with tree-based algorithms can be estimated as $2(\log(p)+k)(\alpha + m\frac{m}{2k}\beta)$ where a message is divided into k chunks to communicate in a pipeline [42].

Table 3: 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	$D\sum_{l=1}^{G}\left(FW_l + BW_l\right) + \frac{D}{B}\sum_{l=1}^{G}(WU_l)$	0	$\gamma\delta\sum_{l=1}^{G}\left(2B(\lvert x_l\rvert + \lvert y_l\rvert) + 2\lvert w_l\rvert + \lvert bi_l\rvert\right)$	$p = 1$
Data	$\frac{D}{p}\sum_{l=1}^{G}(FW_l + BW_l) + \frac{D}{B}\sum_{l=1}^{G}(WU_l)$	$2\frac{D}{B}(p-1)\left(\alpha + \frac{\sum_{l=1}^{G}\lvert w_l\rvert}{p}\delta\beta\right)$	$\gamma\delta\sum_{l=1}^{G}\left(\frac{2B}{p}(\lvert x_l\rvert + \lvert y_l\rvert) + 2\lvert w_l\rvert + \lvert bi_l\rvert\right)$	$p \leq B$
Spatial	$\frac{D}{p}\sum_{l=1}^{G}\left(FW_l + BW_l\right) + \frac{D}{B}\sum_{l=1}^{G}(WU_l)$	$2\frac{D}{B}\Big((p-1)(\alpha + \frac{\sum_{l=1}^{G}\lvert w_l\rvert}{p}\delta\beta) + \sum_{l=1}^{G}\left(2\alpha + B(\text{halo}(\lvert x_l\rvert) + \text{halo}(\lvert\frac{dL}{dy_l}\rvert))\delta\beta\right)\Big)$	$\gamma\delta\sum_{l=1}^{G}\left(2B\frac{(\lvert x_l\rvert+\lvert y_l\rvert)}{p} + 2\lvert w_l\rvert + \lvert bi_l\rvert\right)$	$p = pw \times ph \leq \min_{l=1}^{G}(W_l \times H_l)$
Layer (Pipeline)	$\frac{D(p+S-1)}{S}\Big(\max_{i=1}^{p}(FW_{G_i}) + \max_{i=1}^{p}(BW_{G_i})\Big) + \max_{i=1}^{p}(WU_{G_i})\Big)$	$2\frac{D(p+S-2)}{B}\left(\max_{i=1}^{p-1}\left(\alpha + \frac{B}{S}\lvert y_{G_i}\rvert\delta\beta\right)\right)$	$\gamma\delta\max_{i=1}^{p}\left(\sum_{l=1}^{G_i}(2B(\lvert x_l\rvert + \lvert y_l\rvert) + 2\lvert w_l\rvert + \lvert bi_l\rvert)\right)$	$p \leq G$
Filter	$\frac{D}{p}\sum_{l=1}^{G}\left(FW_l + BW_l\right) + \frac{D}{Bp}\sum_{l=1}^{G}(WU_l)$	$3\frac{D}{B}(p-1)\sum_{l=1}^{G-1}(\alpha + \frac{B\lvert y_l\rvert}{p}\delta\beta)$	$\gamma\delta\sum_{l=1}^{G}\left(2B(\lvert x_l\rvert + \lvert y_l\rvert) + \frac{2\lvert w_l\rvert}{p} + \lvert bi_l\rvert\right)$	$p \leq \min_{l=1}^{G}(F_l)$
Channel	$\frac{D}{p}\sum_{l=1}^{G}\left(FW_l + BW_l\right) + \frac{D}{Bp}\sum_{l=1}^{G}(WU_l)$	$3\frac{D}{B}(p-1)\sum_{l=1}^{G-1}(\alpha + \frac{B\lvert y_l\rvert}{p}\delta\beta)$	$\gamma\delta\sum_{l=1}^{G}\left(2B(\lvert x_l\rvert + \lvert y_l\rvert) + \frac{2\lvert w_l\rvert}{p} + \lvert bi_l\rvert\right)$	$p \leq \min_{l=1}^{G}(C_l)$
Data + Filter	$\frac{D}{p}\sum_{l=1}^{G}\left(FW_l + BW_l\right) + \frac{D}{Bp^2}\sum_{l=1}^{G}(WU_l)$	$3\frac{D}{B}(p2-1)\sum_{l=1}^{G-1}(\alpha + \frac{B\lvert y_l\rvert}{p}\delta\beta) + 2\frac{D}{B}(p1-1)(\alpha + \frac{\sum_{l=1}^{G}\lvert w_l\rvert}{p}\delta\beta)$	$\gamma\delta\sum_{l=1}^{G}\left(\frac{2B(\lvert x_l\rvert+\lvert y_l\rvert)}{p1} + \frac{2\lvert w_l\rvert}{p2} + \lvert bi_l\rvert\right)$	$p = p1 \times p2 \leq B \times \min_{l=1}^{G}(F_l)$

link at each step of collective communications [23]. In our analytical model, we only consider the self-contention caused by all the communication flows of the training process itself, e.g., a link is shared between different groups in hybrid parallelism strategies. The contention coefficient can be estimated analytically by using dynamic contention graphs [30]. It is important to note that we do not intend to model the contention caused by congested networks due to a large number of applications running at the same time in a shared system. Such kind of external contention affects all parallelism strategies and do not reflect the baseline fundamental performance of each parallelism strategy. In addition, the baseline performance predicted by our analytical model can be complemented with a congestion impact factor, which can be empirically estimated as in [7], in order to predict the real-world performance in production environments.

4.4 Empirical Parametrization

As mentioned earlier, we rely on a hybrid of analytical modeling and empirical parametrization for ParaDL. To reduce the impact of noise associated with black-box empirical modeling [8], we segment the experiments used to inspect the target parameters. We are thus able to distinguish between effects of noise on the measurements and actual runtime change because of parameter influence. The empirical parameters are (as defined in Table 2):

- **Computation parameters** (FW_l, BW_l, and WU_l): It is important to note that processors, CPUs and GPUs, rarely perform close to their peak performance. 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. Furthermore, the empirical compute time, per a given layer on a given processor, is available in DL databases of models [28].
- **Communication parameters** (α and β): The interconnect hierarchy of modern computing systems, the algorithms used by communication libraries, and the communication technologies (such as GPUDirect [36]) may lead to differences in the latency and bandwidth factors α and β. Thus, we empirically measure the communication time of collective communication patterns, such as Allreduce, with different message size, number of involved processing elements on a specific computing system. Those empirical measurements can be derived from well-known tools for performance of systems, e.g., OSU Micro-Benchmarks or NCCL-test [27]. We then use those benchmark results to interpolate α and β. It is important to note that α and β become different when changing the number of processing elements in a hierarchical computing architecture, e.g. intra-node, intra-rack and, inter-rack communication.

Table 4: Implementation Overview(✓: customized; - : untouched)

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

It is important to emphasize that the empirical parameters in our model are invariant to the implementation of the parallelism strategies, i.e., values of empirical parameters could change when moving from one framework to another, yet values of the analytical parameters would not. Finally, to simplify the portability of ParaDL between different frameworks and systems, we include the following with the ParaDL utility: a) detailed instructions of using the benchmarks used for gathering the empirical parameters we use, and b) pointers to DL model and layers databases from which the user could get empirical breakdown of compute and memory requirement at the granularity of layers.

4.5 Implementation

4.5.1 Implementation Details. We implement data, channel, filter, spatial and hybrid parallelism strategies using ChainerMN [1] for distributed execution. Although ChainerMN provides a built-in implementation for data parallelism and some minimum 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, TensorFlow, and others). Substantial engineering effort was required to modify and extend the existing implementation and create **ChainerMNX** to support all forms of parallelism. This extension included modifying existing communicators meant for data parallelism to support hybrid parallelism. We also extended existing convolutional layers to support filter/channel/spatial convolutions[5]. We mark our implementation for each type of targeted layers and parallel strategies in Table 4.

[5]The code is publicly available here https://github.com/ankahira/chainermnx

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 final convolution layer (before a fully-connected layer) in VGG16, ResNet-50 and ResNet-152. 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 for ImageNet. In this work we implement the channel parallelism from the second layer. For hybrid strategies such as Data+Filter (or Data+Spatial), we map the data parallelism inter-node. This implementation is also used by Dryden et al. [14]. We leverage ChainerMN with MPI support for both inter-node and intra-node communication. To perform an Allreduce and update the gradients in hybrid strategies, we use ChainerMN's multi-node optimizer which wraps an optimizer and performs an Allreduce before updating the gradients. For the Data+Spatial parallelism, we perform a reduce inside each node to a leader GPU, then perform an Allreduce between the leaders. These two Allreduce involve different parallelization techniques (i.e., spatial in local and data in global). For the Data+Filter parallelism, we perform a segmented Allreduce, e.g., disjoint subsets of GPU run Allreduces on different sets of the weights, i.e., number of subsets equals to number of GPUs per node.

4.5.2 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. Note that these new implementations change only the decomposition of the tensors, and do not change any operator or hyper-paramaters that have an impact on accuracy.

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 [44, 55]. More specifically, for the data parallelism strategy, the typical implementations of batch normalization in commonly used frameworks such as Caffe, PyTorch, TensorFlow are all unsynchronized. This implementation leads to data being only normalized within each PE, separately. In typical 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 [55], which requires a communication overhead for computing the global mini-batch mean. For the 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 [13]. More specifically, BN is typically computed locally on each PE on its own portion of the spatially partitioned data.

Table 5: Models and Datasets Used in Experiments

Model	Dataset	#Samples (Size)	# Param.	#Layers
ResNet-50 [16]	ImageNet [41]	1.28M (3×226^2)	$\approx$ 25M	50
ResNet-152 [16]			$\approx$ 58M	152
VGG16 [47]			$\approx$ 169M	38
CosmoFlow [31]	CosmoFlow [33]	1584 (4×256^3)	$\approx$ 2M	20

In the 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 sends the result to other PEs. Alternatively, each node could redundantly compute the BN layer (distributed approach). In this work, we use the distributed approach which does not require any communication overhead.

5 EVALUATION

In this section, we describe how we conduct a wide range of experiments to show the accuracy and utility of ParaDL in projecting the performance of distributed training of CNN models under different parallel strategies, including hybrid ones. We compare ParaDL projection results to the empirical measurements on a multi-petaflop HPC system with thousands of GPUs. In addition, we characterize the bottlenecks and limitations of different parallelization strategies, and highlight relevant findings observed with the help of ParaDL.

5.1 Methodology

Selected Models and Datasets: We choose different CNN models and datasets with different characteristics that affect performance and memory requirements. They are summarized in Table 5.

Evaluation Environment: Experiments are performed on a multi-petaflop supercomputer, with two Intel Xeon Gold 6148 Processors and four NVIDIA Tesla V100 GPUs (16GB of memory per GPU) on each compute node. The GPUs are connected intra-node to the CPUs by PLX switches and PCIe Gen3 x16 links (16 GBps), and together by NVLink (20 GBps). 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 (two InfiniBand EDR, e.g., 12.5 GBps, per compute node and 17 compute nodes per rack).

Configurations of Experiments: We perform the experiments of the parallel strategies using the framework Chainer [48] (v7.0.0), ChainerMN [1] (the multi-node varient of Chainer), and CUDA (v10.0). We also use the PyTorch (v1.5) implementation for the pipeline strategy [22]. We implement all communication functions based on Nvidia's NCCL library [35] (v2.4.8.1). The exceptions are at which we use MPI (OpenMPI v2.1.6): a) the halo exchange of the spatial strategy since P2P communication interfaces are not supported by NCCL[6], and b) the Allgatherv for the spatial strategy since NCCL does not support Allgatherv.

[6]The latest version of NCCL now supports P2P communications.

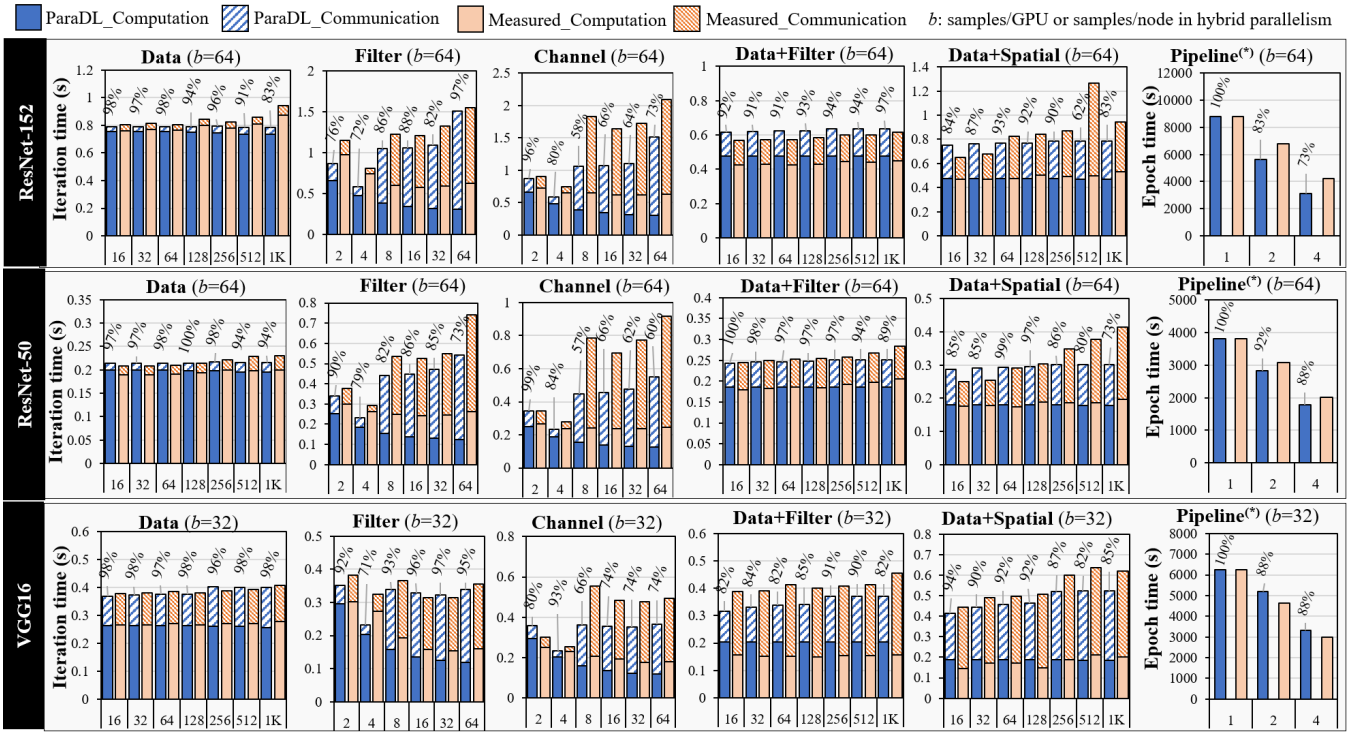


Figure 3: Time breakdown of our analytical model (ParaDL) in comparison with measured runs. The label above each column shows the *projection accuracy*. The x-axis is the number of GPUs. Filter/channel are strong scaling.(*)Values are total time since pipeline parallelism [22] overlaps the computation and communication.

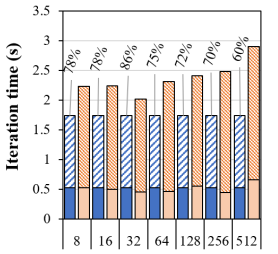


Figure 4: Prediction Accuracy of ParaDL with CosmoFlow for Data+Spatial

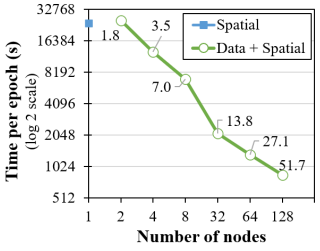


Figure 5: Spatial + data scaling with CosmoFlow. The labels show the speedup ratio of spatial+data over the pure spatial strategy

An important performance factor is efficient device utilization 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 (marked as b in Figure 3). 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 CosmoFlow with spatial strategies, since we use only one sample per node, i.e., 0.25 samples/GPU: we could not have the freedom to tune the parameter b. This is often the case for models using large 3D input datasets (increasingly common in scientific computing), where data parallelism is simply not an option. Given that it was not possible to get the empirical layer by layer time for CosmoFlow running a sequential implementation with the 512^3 data size, we used 256^3 sample sizes and multiplied the computation time by 8. We confirmed with measurements that the strategy was accurate.

5.2 ParaDL's Projection and Accuracy

This section discusses how close is the projection of the ParaDL oracle 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 up to 1024 GPUs). In this comparison, we focus only on the computation and communication time of the main training loop (the most time consuming part) and exclude other times from this study such as I/O staging.

Figure 3 shows the oracle's projections versus the measured runs for different parallel strategies using three different models (a fourth model is shown in Figure 4). We ran all the permutations of possible configurations but plot only some of them because of space limitations. The figure is divided in three rows, one for each CNN model, and six columns, one for each parallelism strategy. The parameter b shows the mini-batch size for each case. As mentioned in Section 5.1, the mini-batch size is set to achieve the

highest device occupancy on GPUs. The x-axis shows the number of GPUs, up to the scaling limit of the specific parallel strategy (e.g., maximum number of filters). More specifically, we scale the tests from 16 to 1024 GPUs for data and hybrid parallelism, from 4 to 64 GPUs for filter/channel parallelism, and up to 4 GPUs for pipeline parallelism. The y-axis shows the iteration time for each case. The iteration time is calculated as an average of 100 iterations excluding the first iteration which normally involves initialization tasks. To get a more detailed analysis, we decompose the execution time into computation and communication. The oracle prediction is shown in blue as stacked bars, i.e., computation+communication, and the measured empirical results are shown in orange. In this figure, we report the best communication times obtained during our experiments, as this represents the peak performance the hardware can deliver and leave aside occasional delays due to external factors such as network congestion coming from other apps, system noise and, overheads due to correctable errors, among others. A detailed analysis on network congestion is given on Section 5.3.1. The labels above each column show the *projection accuracy* in percentage, i.e., 1 - ratio of the absolute value of the difference with respect to the total measured time. Similarly, Figure 4 shows the accuracy for CosmoFlow in the case of Data+Spatial parallelism. Note that the reason CosmoFlow is only run on the Data+Spatial hybrid configurations is because the sample size is so large that it cannot run with any other parallel strategy.

The accuracy of ParaDL predictions for the different parallel strategies are 96.10% for data parallelism, 85.56% for Filter, 73.67% for Channel, 91.43% for Data+Filter, 83.46% for Data+Spatial and 90.22% for pipeline across all CNN models. In general, this represents an overall accuracy of 86.74% for ParaDL, across all parallelism strategies and CNN models, and up to 97.57% for data parallelismon VGG16. CosmoFlow shows an accuracy of 74.14% on average.

It is important to note that the overall average accuracy drops significantly due to some few outliers in which the communication time measured is substantially higher than the prediction from ParaDL. For instance Data+Spatial for ResNet-152 with 512 GPUs shows an accuracy of 62% due to network congestions. Interestingly, the same configuration with 1024 GPUs shows a much higher accuracy (i.e., 83% for Data+Spatial ResNet-152) including the communication part, demonstrating that the ParaDL oracle is highly accurate, even at large scale (i.e., 1024 GPUs). Section 5.3.1 includes a detailed analysis of the network congestion leading to the few outliers where the machine was oversubscribed. Note that the communication time of ParaDL for Data+Filter shown in Figure 3 is calculated with a contention penalty coefficient of 2×, e.g., contention caused by two disjoint Allreduces that share the same InfiniBand link for inter-node communication. The high accuracy reported show that our analytical model fits well to the real performance.

5.3 Parallelism Limitations and Bottlenecks

In this section we use a combination of observations from ParaDL projections and empirical results to highlight some important points: (i) inherent to the parallel strategies themselves (limitations), and (ii) those caused by other components such as the framework implementation or system architecture (bottlenecks). This helps users in avoiding these limitations, and framework programmers in prioritizing their efforts for improvements. We group these limitations and bottlenecks into four categories.

5.3.1 ***Communication.*** It is well-known in literature that parallel training introduces communication overhead. Those overheads have different forms and patterns for different parallel strategies. There is the gradient exchange at the end of each iteration in data and spatial parallelism (GE-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).

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. Another point worth mentioning is the hierarchal implementation of Allreduce in hybrid strategies such as Data+Filter (*df*) and Data+Spatial (*ds*). These two types of Allreduce (data and hybrid parallelism) are different as they involve different parallelization techniques. In *ds*, 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 for Allreduce is more than 2× as those of data. Alternative ways to address this issue are to use multiple leaders instead of only one [34] or to use segmented allreduces, i.e., smaller, concurrent allreduces among disjoint sets of GPUs [14]. We use the former strategy for *ds* and the latter for *df*. These methods are not trivial to implement and they require significant engineering effort. ParaDL has proven accurate at modeling those communications and can be used to choose an implementation strategy based on ParaDL's projected communication times.

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 × batch size, i.e., $O(B\sum_{l=1}^{G}|y_l|)$, 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 $\geq$ 32 samples, the communication time of filter/channel is larger than that of data parallelism (See Figure 3). Note that because this communication overhead is attributed to the forward and backward phases, Allreduce optimization techniques 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 [46] or using a smaller segmented Reduce-Scatter [14].

Peer to Peer communication: The halo exchange in spatial parallelism and the activation passing between composite layers in pipeline are performed in a P2P fashion, which are expected to have small communication times. However, ParaDL shows that the communication time of FB-Halo is non-trivial and this was confirmed by the empirical results. For example, in ResNet-50, 128 GPUs, the time of FB-Halo is approximately 60% of the gradient exchange Allreduce, which is substantially higher than initially expected. This bottleneck appears because the framework uses the MPI library instead of NCCL (NCCL allows GPUs to communicate

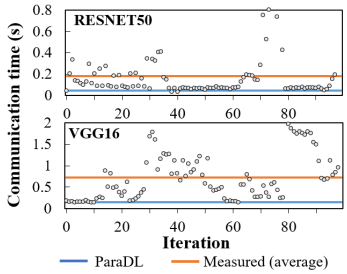


Figure 6: Network congestion of ResNet-50, 512 GPUs, data parallelism (upper) and VGG16, 64 GPUs, filter parallelism (lower).

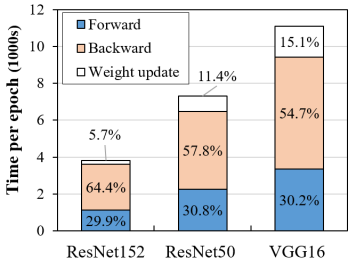


Figure 7: Computation time per epoch with PyTorch. Weight update is not trivial in large models and dataset.

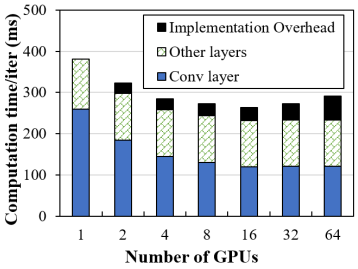


Figure 8: Computation breakdown of filter parallelism, ResNet-50. Implementation of convolution layers does not scale well.

directly instead of via CPU, i.e., GPUDirect). We plugged different network parameters in ParaDL (See Section 4.4) for MPI and NCCL and we confirmed the difference, both theoretically and empirically.

Network Congestion: In our empirical experiments, 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 3, we still observe network congestion when approaching 1K GPUs. We did a detailed analysis for several of the runs. In Figure 6 we show the time for Allreduce communication for data parallelism of ResNet-50 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 (blue line), yet network congestion caused 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.

5.3.2 ***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. One alternative proposed in ZeRO [38] 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 passes. 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 when the input size is 4×512^3. Accordingly, for those kind of models the pipeline strategy would be unfeasible and one has to resort to other parallel strategies (e.g., use Data+Spatial for CosmoFlow as shown in Figure 5). Additionally, since samples are fed 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 [17, 32], which comes with the overhead of recomputing the activations within each partition.

Memory Manager: DL frameworks typically include a memory manager to reduce the overhead of frequent malloc and free. Since GPUs typically operate in asynchronous execution model, there are many CUDA kernels being launched at any given time in training. We observed a disparity between ParaDL and measured performance that could be attributed to stalling kernels. Upon inspection, we observed kernels that were launched asynchronously often stall when requesting memory (in, Chainer as well as PyTorch). The launched asynchronously launched kernels waiting for memory to be available leads to heavy fluctuations in performance. Guided by ParaDL to identify the fine-grained location of the performance gap, we confirmed, for instance, that the implementation of Data+Spatial parallelism of VGG16 (64 GPUs) could avoid out-of-memory issues at the cost of a performance degradation of 1.5×.

5.3.3 ***Computation.*** We highlight the following limitations.

Weight update: most compute time in training typically goes 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 VGG16 (shown in Figure 7). Larger DL models, using ADAM optimizer in specific, may need a higher computation time for weight. Especially, large Transformer based models report up to 45% time on weight update and more than 60% extra memory requirements since ADAM requires four variables per weight [49]. One alternative to address this is to shard the weight update among GPUs across iterations, and Allgather the weights before forward/backward passes [52].

Workload Balancing: Pipeline can outperform data parallelism with less communication by using P2P 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 (e.g. networks with non-linear connections). To further improve load balancing, a straight forward approach is to use data parallelism inside a stage, i.e., hybrid of pipeline plus data [32].

Computation Redundancy: This section discusses limits that could arise from computational redundancy that is introduced for different parallel strategies. Using ParaDL we found out that there was a gap between analytical result and the measured time in filter/channel. Looking in detail, we saw that this was an implementation issue in the framework including two factors i) the convolution layer does not always scale as expected and ii) the computation overhead, such as split/concat, is non-trivial. These non-trivial implementation overheads are shown in Figure 8.

5.3.4 ***Scaling limitation.*** When scaling, there is a limit on the number of GPUs for each of the model parallel strategies (last column of Table 3). For example, p can not exceed the minimum number of filters of a layer in the model, i.e., 64 in the case of VGG16 and ResNet-50 with filter parallelism. Hybrid approaches have a better scaling than those of pure model parallelisms. For example, as shown in Figure 5, 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).

5.4 Other Observations

This section discusses a few more points related to CNN training.

5.4.1 The Rise of Hybrid Parallelism. As mentioned, each of four basic parallelism strategies has its own limitations. Using the hybrid strategies (Data+Model) helps to break/mitigate those limitations, e.g., memory issue of data parallelism and communication and scaling limitation of model parallelism (Section 5.3). As more datasets from the HPC field start to be trained by DL frameworks, this type of hybrid parallel strategies will become increasingly relevant because data parallelism will simply be not enough, as shown in the case of CosmoFlow and its good scaling with *ds* in Figure 5. In addition, hybrid strategies may drive to a better solution in terms of performance. In accordance with other recent reports [14], there are cases where data+filter (*df*) hybrid can outperform data parallelism at large scale, as we also observed in some of our experiments (we also noticed scenarios where pipeline outperforms data parallelism).

5.4.2 Distributed Inference. Inference at large scale is becoming increasingly demanded, given that for large models the inference would also be distributed [4]. 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 [11]). Some of the limitations and bottlenecks of distributed training discussed previously also appear in distributed inference (See lines marked with **Y** in column **I** of Table 6).

6 CONCLUSION

We propose an analytical model for characterizing and identifying the best technique of different parallel strategies for CNN distributed 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. The results demonstrate the accuracy of ParaDL, as high as 97.57% , and 86.74% on average accuracy across all parallel strategies on multiple CNN models and datasets on up to 1K GPUs.

The analytical model helped us uncover limitations and bottlenecks of parallel training of CNNs (summary in Table 6). We analytically identify different bottlenecks that can appear in different parallel strategies due to communication patterns that compensate for different ways to split the tensors, and confirm those predictions empirically. Finally, we identify memory and computational pressure that arises from different redundancies in different parallel strategies.

Table 6: Summary of detected limitations (L) and bottlenecks (B) with the related training phases (✓) and components (●). Those limitations/bottlenecks may appear (Y/N) in distributed inference (I). Related parallel strategies (×): d-data, s-spatial, p-pipeline, f/c-filter and channel, df-hybrid Data+Filter, ds-hybrid Data+Spatial. FR-Framework, SY-System. 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.

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
Memory Capacity	B	×	×	×	×	×	×	○	●	✓	✓	✓	✓	Y	Memory Redundancy
	B	×	×	×	×	×	×	●	○	✓	✓	✓	✓	Y	Memory Stalling
Comput-ation	L	×	×	×	×	×	×	○	○	-	-	-	✓	N	Weight Update
	L	-	-	×	-	-	-	○	○	-	✓	-	✓	Y	Workload Balancing
	B	-	-	-	×	×	-	●	○	-	✓	-	✓	Y	Comp. Redundancy
Scaling	L	×	×	×	×	×	×	○	○	✓	✓	✓	✓	Y	Number of PEs

ACKNOWLEDGMENT

The project that gave rise to these results received the support of a fellowship from the "la Caixa" Foundation (ID **100010434**). The fellowship code is **LCF/BQ/DI17/11620059**.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. **713673**.

The Eurolab4HPC project has received funding from the European Union Horizon 2020 Framework Programme (H2020-EU.1.2.2. - FET Proactive) under grant agreement number **800962**

This work was supported by JST, ACT-X Grant Number JPMJAX190C, Japan; by JST, PRESTO Grant Number JPMJPR20MA, Japan.

REFERENCES

[1] Takuya Akiba et al. 2017. ChainerMN: Scalable Distributed Deep Learning Framework. In *Workshop on ML Systems in NIPS.*

[2] Mohammadreza Bayatpour et al. 2017. Scalable Reduction Collectives with Data Partitioning-based Multi-leader Design *(SC '17).* Article 64, 11 pages.

[3] Tal Ben-Nun et al. 2018. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. *CoRR* abs/1802.09941 (2018). arXiv:1802.09941

[4] Tom B. Brown et al. 2020. Language Models are Few-Shot Learners. *arXiv preprint arXiv:2005.14165* (2020). arXiv:2005.14165 [cs.CL]

[5] Adrián Castelló et al. 2019. Analysis of Model Parallelism for Distributed Neural Networks *(EuroMPI '19).* Article 7, 10 pages.

[6] Tianqi Chen et al. 2016. Training Deep Nets with Sublinear Memory Cost. *ArXiv* abs/1604.06174 (2016).

[7] Sudheer Chunduri et al. 2019. GPCNeT: Designing a Benchmark Suite for Inducing and Measuring Contention in HPC Networks *(SC '19).* Article 42, 33 pages.

[8] Marcin Copik et al. 2020. Extracting Clean Performance Models from Tainted Programs. arXiv:2012.15592 [cs.DC]

[9] Jeffrey Dean et al. 2012. Large Scale Distributed Deep Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems* (Lake Tahoe, Nevada). 1223–1231.

[10] Jens Domke et al. 2019. HyperX Topology: First at-Scale Implementation and Comparison to the Fat-Tree *(SC '19)*. Article 40, 23 pages.

[11] Ge Dong et al. 2020. Fully Convolutional Spatio-Temporal Models for Representation Learning in Plasma Science. *arXiv preprint arXiv:2007.10468* (2020). arXiv:2007.10468 [physics.comp-ph]

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

[13] N. Dryden et al. [n.d.]. Improving Strong-Scaling of CNN Training by Exploiting Finer-Grained Parallelism. In *IPDPS 2019*. 210–220.

[14] Nikoli Dryden et al. 2019. Channel and Filter Parallelism for Large-Scale CNN Training *(SC '19)*. Article 46, 13 pages.

[15] Amir Gholami et al. 2017. Integrated model, batch and domain parallelism in training neural networks. *arXiv preprint arXiv:1712.04432* (2017).

[16] K. He et al. 2016. Deep Residual Learning for Image Recognition. In *CVPR*. 770–778.

[17] Yanping Huang et al. 2018. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *CoRR* abs/1811.06965 (2018). arXiv:1811.06965

[18] Xianyan Jia et al. 2018. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. *CoRR* abs/1807.11205 (2018). arXiv:1807.11205

[19] Zhihao Jia et al. 2018. Exploring hidden dimensions in parallelizing convolutional neural networks. *arXiv preprint arXiv:1802.04924* (2018).

[20] Hai Jin et al. 2018. Layer-Centric Memory Reuse and Data Migration for Extreme-Scale Deep Learning on Many-Core Architectures. *ACM Trans. Archit. Code Optim.* 15, 3, Article 37 (Sept. 2018). https://doi.org/10.1145/3243904

[21] Albert Kahira et al. 2018. Training Deep Neural Networks with Low Precision Input Data: A Hurricane Prediction Case Study. In *High Performance Computing*. Springer International Publishing, Cham, 562–569.

[22] Chiheon Kim et al. 2020. torchgpipe: On-the-fly Pipeline Parallelism for Training Giant Models. *arXiv preprint arXiv:2004.09910* (2020). arXiv:2004.09910 [cs.DC]

[23] Sang Cheol Kim et al. 2001. Measurement and Prediction of Communication Delays in Myrinet Networks. *J. Parallel and Distrib. Comput.* 61, 11 (2001), 1692 – 1704.

[24] Alexander Kolesnikov et al. 2019. Big Transfer (BiT): General Visual Representation Learning. *arXiv preprint arXiv:1912.11370* (2019). arXiv:1912.11370 [cs.CV]

[25] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

[26] Thorsten Kurth et al. 2018. Exascale Deep Learning for Climate Analytics *(SC '18)*. 51:1–51:12.

[27] Ang Li et al. 2019. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *CoRR* abs/1903.04611 (2019). arXiv:1903.04611

[28] Cheng Li et al. 2019. Benanza: Automatic μBenchmark Generation to Compute "Lower-bound" Latency and Inform Optimizations of Deep Learning Models on GPUs. *ArXiv* abs/1911.06922 (2019).

[29] Sangkug Lym et al. 2019. PruneTrain: Fast Neural Network Training by Dynamic Sparse Model Reconfiguration *(SC '19)*. Article 36, 13 pages.

[30] Maxim Martinasso et al. 2011. A Contention-Aware Performance Model for HPC-Based Networks: A Case Study of the InfiniBand Network. In *Euro-Par 2011 Parallel Processing*.

[31] Amrita Mathuriya et al. 2018. CosmoFlow: Using Deep Learning to Learn the Universe at Scale *(SC '18)*. Article 65, 11 pages.

[32] Deepak Narayanan et al. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training *(SOSP '19)*. 1–15.

[33] National Energy Research Scientific Computing Center. [n.d.]. CosmoFlow datasets. https://portal.nersc.gov/project/m3363/. [15 January 2020].

[34] Truong Thao Nguyen et al. 2018. Hierarchical Distributed-Memory Multi-Leader MPI-Allreduce for Deep Learning Workloads *(CANDAR18)*. 216–222.

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

[36] NVIDIA. [n.d.]. GPUDirect. https://developer.nvidia.com/gpudirect. [21 April 2020].

[37] J. Gregory Pauloski et al. 2020. Convolutional Neural Network Training with Distributed K-FAC. *arXiv preprint arXiv:2007.00784* (2020).

[38] Samyam Rajbhandari et al. 2019. ZeRO: Memory Optimization Towards Training A Trillion Parameter Models. *ArXiv* abs/1910.02054 (2019).

[39] Cedric Renggli et al. 2019. SparCML: High-Performance Sparse Communication for Machine Learning *(SC '19)*. Article 11, 15 pages.

[40] Juan A. Rico-Gallego et al. 2019. A Survey of Communication Performance Models for High-Performance Computing. *ACM Comput. Surv.* 51, 6, Article 126 (Jan. 2019), 36 pages.

[41] Olga Russakovsky et al. 2015. Imagenet large scale visual recognition challenge. *International journal of computer vision* 115, 3 (2015), 211–252.

[42] Peter Sanders et al. 2009. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Comput.* 35, 12 (2009), 581 – 594.

[43] Frank Seide et al. 2014. 1-Bit Stochastic Gradient Descent and Application to Data-Parallel Distributed Training of Speech DNNs *(Interspeech 2014)*.

[44] Ioffe Sergey et al. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR* abs/1502.03167 (2015). arXiv:1502.03167

[45] Yelong Shen et al. 2014. Learning Semantic Representations Using Convolutional Neural Networks for Web Search *(WWW '14 Companion)*. Association for Computing Machinery, New York, NY, USA, 373–374.

[46] Mohammad Shoeybi et al. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *ArXiv* abs/1909.08053 (2019).

[47] Karen Simonyan et al. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR 2015*.

[48] Seiya Tokui et al. 2015. Chainer: a Next-Generation Open Source Framework for Deep Learning. In *Proceedings of LearningSys in NIPS*.

[49] Mohamed Wahib et al. 2020. Scaling Distributed Deep Learning Workloads beyond the Memory Capacity with KARMA. *arXiv preprint arXiv:2008.11421* (2020).

[50] Izhar Wallach et al. 2015. AtomNet: A Deep Convolutional Neural Network for Bioactivity Prediction in Structure-based Drug Discovery. arXiv:1510.02855 [cs.LG]

[51] Yuxin Wu and Kaiming He. 2018. Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*. 3–19.

[52] Yuanzhong Xu et al. 2020. Automatic Cross-Replica Sharding of Weight Update in Data-Parallel Training. *arXiv preprint arXiv:2004.13336* (2020). arXiv:2004.13336 [cs.DC]

[53] Masafumi Yamazaki et al. 2019. Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds. *CoRR* abs/1903.12650 (2019). arXiv:1903.12650

[54] Yang You et al. 2018. ImageNet Training in Minutes. In *Proceedings of the 47th International Conference on Parallel Processing* (Eugene, OR, USA) *(ICPP 2018)*. 1:1–1:10.

[55] Hang Zhang et al. 2018. Context Encoding for Semantic Segmentation. In *CVPR2018*.

[56] Jia Zhihao et al. 2018. Beyond Data and Model Parallelism for Deep Neural Networks. *CoRR* abs/1807.05358 (2018). arXiv:1807.05358

A APPENDIX

A.1 Performance and Memory Analysis

The training time of one epoch in the sequential implementation (serial) of a CNN 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.

A.1.1 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 needs memory to store its input $B'|x_l|$, activation $B'|y_l|$, weights $|w_l|$, bias $|bi_l|$, the gradients $B'|\frac{dL}{dx_l}|$, $B'|\frac{dL}{dy_l}|$, and $|\frac{dL}{dw_l}|$. Overall, if each item of the input, activation, weight and gradients are stored in δ bytes, the maximum required memory at one PE is:

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

A.1.2 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:

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

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.

A.1.3 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:

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

A.1.4 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)[7] 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:

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

A.1.5 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

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

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

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:

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

[7] 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.

A.1.6 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:

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

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 3.