HyPar-Flow: Exploiting MPI and Keras for Scalable Hybrid-Parallel DNN Training with TensorFlow

Ammar Ahmad Awan(✉), Arpan Jain, Quentin Anthony, Hari Subramoni, and Dhabaleswar K. Panda

The Ohio State University, Columbus, OH 43210, USA
{awan.10,jain.575,anthony.301,subramoni.1,panda.2}@osu.edu

Abstract. To reduce the training time of large-scale Deep Neural Networks (DNNs), Deep Learning (DL) scientists have started to explore parallelization strategies like data-parallelism, model-parallelism, and hybrid-parallelism. While data-parallelism has been extensively studied and developed, several problems exist in realizing model-parallelism and hybrid-parallelism efficiently. Four major problems we focus on are: 1) defining a notion of a distributed model across processes, 2) implementing forward/back-propagation across process boundaries that requires explicit communication, 3) obtaining parallel speedup on an inherently sequential task, and 4) achieving scalability without losing out on a model's accuracy. To address these problems, we create **HyPar-Flow**—a model-size and model-type agnostic, scalable, practical, and user-transparent system for hybrid-parallel training by exploiting MPI, Keras, and TensorFlow. HyPar-Flow provides a single API that can be used to perform data, model, and hybrid parallel training of any Keras model at scale. We create an internal distributed representation of the user-provided Keras model, utilize TF's Eager execution features for distributed forward/back-propagation across processes, exploit pipelining to improve performance and leverage efficient MPI primitives for scalable communication. Between model partitions, we use *send* and *recv* to exchange layer-data/partial-errors while *allreduce* is used to accumulate/average gradients across model replicas. Beyond the design and implementation of HyPar-Flow, we also provide comprehensive correctness and performance results on three state-of-the-art HPC systems including TACC *Frontera* (#5 on Top500.org). For ResNet-1001, an ultra-deep model, HyPar-Flow provides: 1) Up to 1.6× speedup over Horovod-based data-parallel training, 2) 110× speedup over single-node on 128 Stampede2 nodes, and 3) 481× speedup over single-node on 512 Frontera nodes.

Keywords: Hybrid parallelism · Model parallelism · Keras · TensorFlow · MPI · Eager Execution · Deep Learning · DNN training

This research is supported in part by NSF grants #1931537, #1450440, #1664137, #1818253, and XRAC grant #NCR-130002.

© Springer Nature Switzerland AG 2020
P. Sadayappan et al. (Eds.): ISC High Performance 2020, LNCS 12151, pp. 83–103, 2020.
https://doi.org/10.1007/978-3-030-50743-5_5

1 Introduction and Motivation

Recent advances in Machine/Deep Learning (ML/DL) have triggered key success stories in many application domains like Computer Vision, Speech Comprehension and Recognition, and Natural Language Processing. Large-scale Deep Neural Networks (DNNs) are at the core of these state-of-the-art AI technologies and have been the primary drivers of this success. However, training DNNs is a compute-intensive task that can take weeks or months to achieve state-of-the-art prediction capabilities (*accuracy*). These requirements have led researchers to resort to a simple but powerful approach called *data-parallelism* to achieve shorter training times. Various research studies [5,10] have addressed performance improvements for data-parallel training. As a result, production-grade ML/DL software like TensorFlow and PyTorch also provide robust support for data-parallelism.

While data-parallel training offers good performance for models that can completely reside in the memory of a CPU/GPU, it *can not* be used for models larger than the memory available. Larger and deeper models are being built to increase the accuracy of models even further [1,12]. Figure 1 highlights how *memory consumption* due to larger images and DNN depth limits the compute platforms that can be used for training; e.g. ResNet-1k [12] with the smallest possible batch-size of one (a single 224×224 image) needs 16.8 GB memory and thus *cannot* be trained on a 16 GB Pascal GPU. Similarly, ResNet-1k on image size 720×720 needs 153 GB of memory, which makes it out-of-core for most platforms except CPU systems that have 192 GB memory. These *out-of-core* models have triggered the *need for model/hybrid parallelism*.

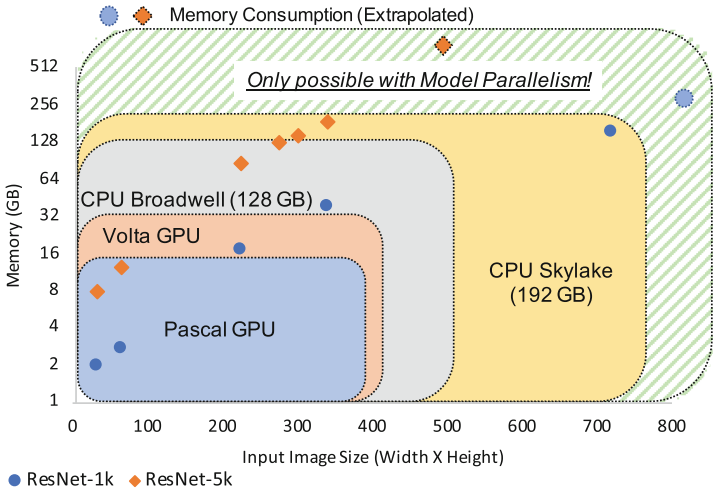


Fig. 1. The need for model/hybrid-parallelism

However, realizing *model-parallelism*—splitting the model (DNN) into multiple partitions—is non-trivial and requires the knowledge of best practices in ML/DL as well as expertise in High Performance Computing (HPC). We note that model-parallelism and layer-parallelism can be considered equivalent terms when the smallest partition of a model is a layer [7,15]. Little exists in the literature about model-parallelism for state-of-the-art DNNs like ResNet(s) on HPC systems. Combining data and model parallelism, also called hybrid-parallelism has received even less attention. Realizing model-parallelism and hybrid-parallelism efficiently is challenging because of *four major problems*: 1) defining a distributed model is necessary but difficult because it requires knowledge of the model as well as of the underlying communication library and the distributed hardware, 2) implementing distributed forward/back-propagation is needed because partitions of the model now reside in different memory spaces and will need explicit communication, 3) obtaining parallel speedup on an inherently sequential task; forward pass followed by a backward pass, and 4) achieving scalability without losing out on a model's accuracy.

Proposed Approach: To address these four problems, we propose HyPar-Flow: a scalable, practical, and user-transparent system for hybrid-parallel training on HPC systems. We offer a simple interface that does not require any model-definition changes and/or manual partitioning of the model. Users provide four inputs: 1) A model defined using the Keras API, 2) Number of model partitions, 3) Number of model replicas, and 4) Strategy (data, model, or hybrid). Unlike existing systems, we design and implement all the cumbersome tasks like splitting the model into partitions, replicating it across processes, pipelining over batch partitions, and realizing communication inside HyPar-Flow. This enables the users to focus on the science of the model instead of system-level problems like the creation of model partitions and replicas, placement of partitions and replicas on cores and nodes, and performing communication between them. HyPar-Flow's simplicity from a user's standpoint and its complexity (hidden from the user) from our implementation's standpoint is shown in Fig. 2.

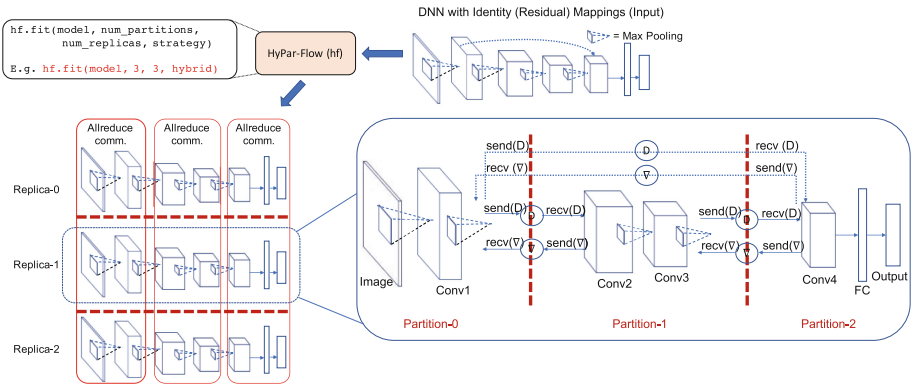


Fig. 2. Proposed user-transparent hybrid-parallel training approach (HyPar-Flow)

1.1 Contributions

From a research and novelty standpoint, our proposed solution is both model-size as well as model-type agnostic. It is also different compared to all existing systems because we focus on high-level and abstract APIs like Keras that are used in practice instead of low-level tensors and matrices, which would be challenging to use for defining state-of-the-art models with hundreds of layers. HyPar-Flow's solution to communication is also novel because it is the first system to exploit standard Message Passing Interface (MPI) primitives for inter-partition and inter-replica communication instead of reinventing single-use libraries. To the best of our knowledge, there are very few studies that focus on hybrid-parallel training of large DNNs; especially using TensorFlow and Keras in a user-transparent manner for HPC environments where MPI is a dominant programming model. We make the following key contributions in this paper:

- Analyze various model-definition APIs and DL frameworks and highlight why *Keras* APIs and custom-built training loops using TensorFlow Eager's *GradientTape* are well suited for realizing user-transparent hybrid-parallelism.
- Propose, design, and implement HyPar-Flow to enable parallel training of any Keras model (with consecutive as well as non-consecutive layer connections [7]) on multiple processes under any parallelization <strategy>, i.e. data, model, and hybrid.
- Thoroughly verify the correctness of the HyPar-Flow framework by training the models to state-of-the-art published accuracy.
- Evaluate HyPar-Flow's performance using a variety of models including VGG-16, ResNet-110, ResNet-1001, and AmoebaNet on three HPC systems
- Report up to 3.1× speedup over sequential training for ResNet-110 and up to 1.6× speedup over data-parallel training for ResNet-1001 on a single node.
- Report 110× speedup over single-node on 128 Stampede2 nodes and 481× speedup over single-node on 512 Frontera nodes for ResNet-1001.

2 The Design Space for Parallel Training Frameworks

Alex Krizhevsky introduced model-parallelism on GPUs in [15] using a single-tower design that used data-parallelism in convolutional layers but model-paralle-lism in fully-connected layers. Simulation-based results about various parallelization strategies are presented in [9]. The LBANN team presented model-parallel solutions including support for spatial convolutions split across nodes in [8]. However, model-parallelism in LBANN is not yet publicly available so we cannot compare its performance with HyPar-Flow. MXNet-MP [2] also offers model-parallelism support but no working examples are available at the time of writing. GPipe [13] enables the training of extremely large models like AmoebaNet [19] on Google TPUs and accelerators. GPipe is publicly available but we found no examples and/or documentation to train models like ResNet(s) with model-parallel support on an HPC system. FlexFlow [14] searches parallelization strategies using simulation algorithms and highlights different dimensions of parallelism in DNNs. FlexFlow uses Legion [6] for communication within

the node and GASNet across nodes. Unfortunately, FlexFlow only works on GPUs so we cannot offer a direct comparison. Also, we were unable to configure FlexFlow for multiple nodes. Mesh-TensorFlow (MTF) [20] is a language for distributed DL with an emphasis on tensors distributed across a processor mesh. MTF only works with the older TF APIs (sessions, graphs, etc.). Furthermore, the level at which MTF distributes work is much lower compared to HyPar-Flow, i.e., tensors vs. layers. Users of MTF need to re-write their entire model to be compatible with MTF APIs. Unlike MTF, HyPar-Flow works on the existing models without requiring any code/model changes. We summarize these related studies on data, model, and hybrid-parallelism and their associated features in Table 1. Out-of-core methods like [4,17] take a different approach to deal with large models, which is not directly comparable to model/hybrid-parallelism. Several data-parallelism only studies have been published that offer speedup over sequential training [3,5,10,18,21]. However, all of these are only limited to models that can fit in the main memory of the GPU/CPU.

Table 1. Features offered by HyPar-flow compared to existing frameworks

Existing and proposed studies	Features and supported platforms					
	User transparent	Speedup over data-parallel	Communicationruntim Runtimee/library	Publicly available MP support	Compatible w/Keras	Compatible w/TF Eager
AlexNet [15,16]	×	✔	CUDA	×	×	×
MXNet-MP [2]	×	Unknown	MPI	✔	✔	×
LBANN [8]	✔	✔	MPI/Aluminum	×	×	×
Mesh TensorFlow [20]	×	✔	MPI	✔	×	×
Gpipe [13]	×	×	gRPC/TF	✔	×	Unknown
PipeDream [11]	×	✔	ZeroMQ	Unknown	×	×
FlexFlow [14]	✔	✔	Legion/GASNet	✔	×	×
Proposed (HyPar-Flow)	✔	✔	MPI	Planned	✔	✔

3 Background

We provide the necessary background in this section.

DNN Training: A DNN consists of different types of *layers* such as convolutions (*conv*), fully-connected or dense (*FC*), pooling, etc. DNNs are usually trained using a labeled dataset. A full pass over this dataset is called an *epoch* of training. Training itself is an iterative process and each iteration happens in two broad phases: 1) Forward pass over all the layers and 2) Back-propagation of loss (or error) in the reverse order. The end goal of DNN training is to obtain a model that has good prediction capabilities (*accuracy*). To reach the desired/target *accuracy* in the fastest possible time, the training process itself needs to be efficient. In this context, the total training time is a product of two metrics:

1) the number of epochs required to reach the target accuracy and 2) the time required for one epoch of training.

Data-Parallelism: In data-parallel training, the complete DNN is replicated across all processes, but the training dataset is partitioned across the processes. Since the model replicas on each of the processes train on different partitions of data, the weights (or parameters) learned are different on each process and thus need to be synchronized among replicas. In most cases, this is done by averaging the *gradients* from all processes. This synchronization is performed by using a collective communication primitive like allreduce or by using parameter servers. The synchronization of weights is done at the end of every batch. This is referred to as *synchronous parallel* in this paper.

Model and Hybrid-Parallelism: Data-parallelism works for models that can fit completely inside the memory of a single GPU/CPU. But as model sizes have grown, model designers have pursued aggressive strategies to make them fit inside a GPU's memory, which is a precious resource even on the latest Volta GPU (32 GB). This problem is less pronounced for CPU-based training as the amount of CPU memory is significantly higher (192 GB) on the latest generation CPUs. Nevertheless, some models can not be trained without splitting the model into partitions; Hence, model-parallelism is a necessity, which also allows the designers to come up with new models without being restricted to any memory limits. The entire model is partitioned and each process is responsible only for part (e.g. a layer or some layers) of the DNN. Model-parallelism can be combined with data-parallelism as well, which we refer to as hybrid-parallelism.

4 Challenges in Designing Model and Hybrid-Parallelism

We expand on *four problems* discussed earlier in Sect. 1 and elaborate specific challenges that need to be addressed for designing a scalable and user-transparent system like HyPar-Flow.

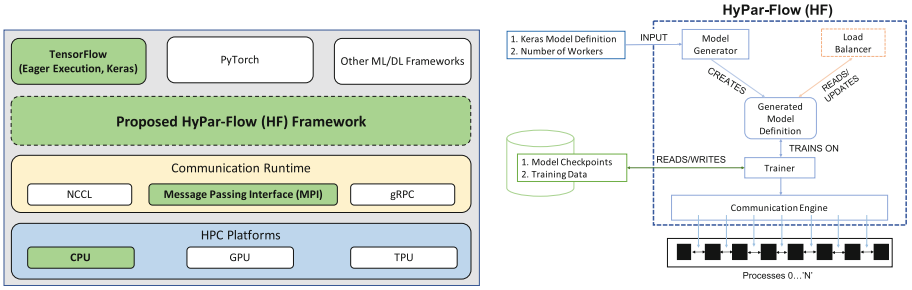
Challenge-1: Model-Definition APIs and Framework-Specific Features To develop a practical system like HyPar-Flow, it is essential that we thoroughly investigate APIs and features of DL frameworks. In this context, the design analysis of execution models like Eager Execution vs. Graph (or Lazy) Execution is fundamental. Similarly, analysis of model definition APIs like TensorFlow Estimators compared to Keras is needed because these will influence the design choices for developing systems like HyPar-Flow. Furthermore, the granularity of interfaces needs to be explored. For instance, using tensors to define a model is very complex compared to using a high-level model API like Keras and ONNX that follow the layer abstraction. Finally, we need to investigate the performance behavior of these interfaces and frameworks. Specific to HyPar-Flow, the main requirement from an API's perspective is to investigate a mechanism that allows us to perform user-transparent model partitioning. Unlike other APIs, Keras seems to provide us this capability via the *tf.keras.Model* interface.

Challenge-2: Communication Between Partitions and Replicas Data-parallelism is easy to implement as no modification is required to the forward pass or the back-propagation of loss (error) in the backward pass. However, for model-parallelism, we need to investigate methods and framework-specific functionalities that enable us to implement the forward and backward pass in a distributed fashion. To realize these, explicit communication is needed between model partitions. For hybrid-parallelism, even deeper investigation is required because communication between model replicas and model partitions needs to be well-coordinated and possibly overlapped. In essence, we need to design a distributed system, which embeds communication primitives like *send*, *recv*, and *allreduce* for exchanging partial error terms, gradients, and/or activations during the forward and backward passes. An additional challenge is to deal with newer DNNs like ResNet(s) [12] as they have evolved from a linear representation to a more complex graph with several types of skip connections (shortcuts) like identity connections, convolution connections, etc. For skip connections, maintaining dependencies for layers as well as for model-partitions is also required to ensure deadlock-free communication across processes.

Challenge-3: Applying HPC Techniques to Improve Performance Even though model-parallelism and hybrid-parallelism look very promising, it is unclear if they can offer performance comparable to data-parallelism. To achieve performance, we need to investigate if applying widely-used and important HPC techniques like 1) efficient placement of processes on CPU cores, 2) pipelining via batch splitting, and 3) overlap of computation and communication can be exploited for improving performance of model-parallel and hybrid-parallel training. Naive model-parallelism will certainly suffer from under-utilization of resources due to stalls caused by the sequential nature of computation in the forward and backward passes.

5 HyPar-Flow: Proposed Architecture and Designs

We propose HyPar-Flow as an abstraction between the high-level ML/DL frameworks like TensorFlow and low-level communication runtimes like MPI as shown in Fig. 3(a). The HyPar-Flow middleware is directly usable by ML/DL applications and no changes are needed to the code or the DL framework. The four major internal components of HyPar-Flow, shown in Fig. 3(b), are 1) Model Generator, 2) Trainer, 3) Communication Engine (CE), and 4) Load Balancer. The subsections that follow provide details of design schemes and strategies for HyPar-Flow and challenges (**C1**–**C3**) addressed by each scheme.



(a) Overview of the Execution Stack (b) Major Components of HyPar-Flow

Fig. 3. HyPar-Flow: a middleware for hybrid-parallel training

5.1 Designing Distributed Model Representation (Address C1)

The *Model Generator* component is responsible for creating an internal representation of a DNN (e.g. a Keras model) suitable for distributed training (Fig. 2). In the standard single-process (sequential) case, all trainable variables (or weights) of a model exist in the address space of a single process, so calling *tape.gradients()* on a *tf.GradientTape* object to get gradients will suffice. However, this is not possible for model-parallel training as trainable variables (weights) are distributed among model-partitions. To deal with this, we first create a local model object on all processes using the *tf.keras.model* API. Next, we identify the layers in the model object that are local to the process. Finally, we create dependency lists that allow us to maintain layer and rank dependencies for each of the local model's layers. These three components define our internal distributed representation of the model. This information is vital for realizing distributed back-propagation (discussed next) as well as for other HyPar-Flow components like the *Trainer* and the *Communication Engine*.

5.2 Implementing Distributed Back-Propagation (Address C1,C2)

Having a distributed model representation is crucial. However, it is only the first step. The biggest challenge for HyPar-Flow and its likes are: "How to train a model that is distributed across process boundaries?". We deal with this challenge inside the *Trainer* component. First, we analyze how training is performed on a standard (non-distributed) Keras model. Broadly, there are two ways to do so: 1) *model.fit(..)* and 2) *model.train_on_batch(..)*. Second, we explore how we can design an API that is very similar to the standard case. To this end, we expose a single *hf.fit(..)* interface that takes parallelization strategy as an argument. The value of the *strategy* argument can be model, data, or hybrid. Third, we design a custom training loop for distributed back-propagation for the model/hybrid parallel case. For data-parallel, it is not needed because the model is replicated on all processes instead of being distributed across processes.

We show a very simple DNN in Fig. 4 to explain back-propagation and highlight what needs to be done for realizing a distributed version. In addition to

Fig. 4, we use Eqs. 1–7 to provide a more detailed explanation. There are three key data elements in DNN Training: 1) The input X, 2) The predicted output Y', and 3) The actual output (or label) Y. The intermediate output from the hidden layer is denoted as V. The difference between Y and Y' is called error or loss labeled as L (Eq. 2).

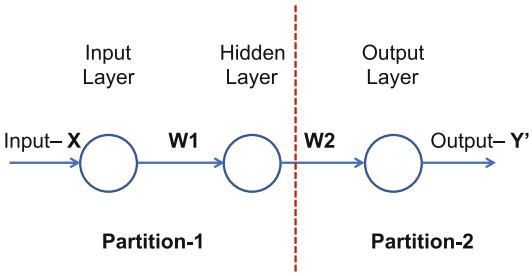


Fig. 4. A neural network with a single hidden layer

$$Y = ActualOutput, Y' = PredictedOutput \tag{1}$$

$$L(Loss) = loss\_function(Y, Y') \tag{2}$$

$$V(HiddenLayer) = W_1(Weight - on - hidden - layer) * X(Input) \tag{3}$$

$$Y'(PredictedOutput) = W_2(Weight - on - output - layer) * V \tag{4}$$

$$D2 = \frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial Y'} * \frac{\partial Y'}{\partial W_2} \tag{5}$$

$$D1 = \frac{\partial L}{\partial W_1} = partial\_error * \frac{\partial V}{\partial W_1} \tag{6}$$

$$partial\_error = \frac{\partial L}{\partial Y'} * \frac{\partial Y'}{\partial V} \tag{7}$$

To realize distributed back-propagation, we need 1) partial derivative (D1) of Loss L with respect to the weight $W1$, and 2) partial derivative (D2) of Loss L with respect to the weight $W2$. The challenge for multi-process case is that the term called "partial error" shown in Eqs. 6 and 7 can only be calculated on *Partition-2* (Fig. 4) as Y' only exists there. To calculate D1, *Partition-1* needs this "partial error" term in addition to D1. Because we rely on accessing gradients using the DL framework's implementation, this scenario poses a fundamental problem. TensorFlow, the candidate framework for this work, does not provide a way to calculate gradients that are not part of a layer. To implement this functionality, we introduce the notion of *grad layer* in HyPar-Flow, which acts as a pseudo-layer inserted before the actual layer on each model-partition. We note that TensorFlow's *GradientTape* cannot be directly used for this case.

Grad layers ensure that we can call *tape.gradients()* on this *grad layer* to calculate the partial errors during back-propagation. Specifically, a *grad layer* is required for each *recv* operation so that partial error can be calculated for each preceding partition's input. A call to tape.gradients() will return a list that contains gradients as well as partial errors. The list is then used to update the model by calling *optimizer.apply_gradients()*.

We note that there is no need to implement distributed back-propagation for the data-parallel case as each model-replica is independently performing the Forward and Backward pass. The gradients are only synchronized (averaged) at the end of the Backward pass (back-propagation) using *allreduce* to update the model weights in a single step.

5.3 Realizing Inter-Partition/-Replica Comm. (Address C2,C3)

In Sects. 5.1 and 5.2, we discussed how the distributed model definition is generated and how back-propagation can be implemented for a model that is distributed across processes. However, *Trainer* and *Model Generator* only provide an infrastructure for distributed training. The actual communication of various types of data is realized in HyPar-Flow's *Communication Engine (CE)*. The CE is a light-weight abstraction for internal usage and it provides four simple APIs: 1) send, 2) recv, 3) broadcast and 4) allreduce.

HyPar-Flow CE Basic Design: For pure data-parallelism, we only need to use allreduce. However, for model-parallelism, we also need to use point-to-point communication between model-partitions. In the forward pass, the send/recv combination is used to propagate *partial predictions* from each partition to the next partition starting at *Layer 1*. On the other hand, send/recv is used to back-propagate the *loss* and *partial-errors* from one partition to the other starting at *Layer N*. Finally, for hybrid-parallelism, we need to introduce allreduce to accumulate (average) the gradients across model replicas. We note that this is different from the usage of allreduce in pure data-parallelism because in this case, the model itself is distributed across different partitions so allreduce cannot be called directly on all processes. One option is to perform another p2p communication between model replicas for gradient exchange. The other option is to exploit the concept of MPI communicators. We choose the latter one because of its simplicity as well as the fact the MPI vendors have spent considerable efforts to optimize the allreduce collective for a long time. To realize this, we consider the same model-partition for all model-replicas to form the *Allreduce communicator*. Because we only need to accumulate the gradients local to a partition across all replicas, allreduce called on this communicator will suffice. Please refer back to Fig. 2 (Sect. 1) for a graphical illustration of this scheme.

HyPar-Flow CE Advanced Design: The basic CE design described above works but does not offer good performance. To push the envelope of performance further, we investigate two HPC optimizations: 1) we explore if the overlap of computation and communication can be exploited for all three parallelization

strategies and 2) we investigate if pipelining can help overcome some of the limitations that arise due to the sequential nature of the forward/backward passes. Finally, we also handle some advanced cases for models with non-consecutive layer connections (e.g. ResNet(s)), which can lead to deadlocks.

Exploiting Overlap of Computation and Communication: To achieve near-linear speedups for data-parallelism, the overlap of computation (forward/backward) and communication (allreduce) has proven to be an excellent choice. Horovod, a popular data-parallelism middleware, provides this support so we simply use it inside HyPar-Flow for pure data-parallelism. However, for hybrid-parallelism, we design a different scheme. We create one MPI communicator per model partition whereas the size of each communicator will be equal to the number of model-replicas. This design allows us to overlap the allreduce operation with the computation of other partitions on the same node. An example scenario clarifies this further: if we split the model across 48 partitions, then we will use 48 allreduce operations (one for each model-partition) to get optimal performance. This design allows us to overlap the allreduce operation with the computation of other partitions on the same node.

Exploiting Pipeline Stages within Each Minibatch: Because DNN training is inherently sequential, i.e., the computation of each layer is dependent on the completion of the previous layer. This is true for the forward pass, as well as for the backward pass. To overcome this performance limitation, we exploit a standard technique called pipelining. The observation is that DNN training is done on batches (or mini-batches) of data. This offers an opportunity for pipelining as a training step on samples within the batch is parallelizable. Theoretically, the number of pipeline stages can be varied from 1 all the way to batch size. This requires tuning or a heuristic and will vary according to the model and the underlying system. Based on hundreds of experiments we performed for HyPar-Flow, we derive a simple heuristic: use the largest possible number for pipeline stages and decrease it by a factor of two. In most cases, we observed that $num\_pipeline\_stages = batch\_size$ provides the best performance.

Special Handling for Models with Skip Connections: Figure 5 shows a non-consecutive model with skip connections that requires communication 1) between adjacent model-partitions for boundary layers and 2) non-adjacent model-partitions for the skip connections. To handle communication dependencies among layers for each model-partition, we create two lists: 1) Forward list and 2) Backward list. Each list is a list of lists to store dependencies between layers as shown in Fig. 5. "F" corresponds to the index of the layer to which the current layer is sending its data and "B" corresponds to the index of the layer from which the current layer is receiving data. An arbitrary sequence of sending and receiving messages may lead to a deadlock. For instance, if *Partition-1* sends the partial predictions to *Partition-3* when *Partition-3* is waiting for predictions from *Partition-2*, a deadlock will occur as *Partition-2* is itself blocked (waiting for results from *Partition-1*). To deal with this, we sort the message sequence according to the ranks so that the partition sends the first message to the partition which has the next layer.

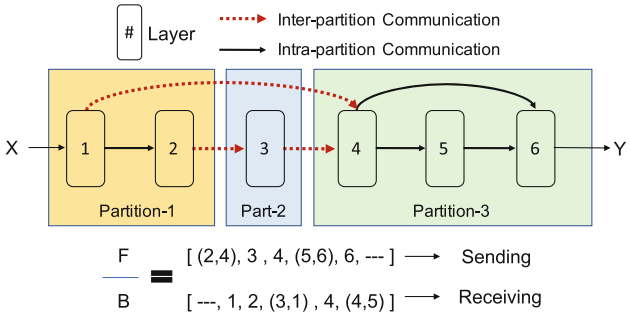


Fig. 5. Avoiding deadlocks for models with non-consecutive connections

5.4 Load Balancer

The models we used did not show any major load imbalance but we plan to design this component in the future to address emerging models from other application areas that require load balancing capabilities from HyPar-Flow.

6 Performance Characterization and Correctness Testing

We have used three HPC systems to evaluate the performance and test the correctness of HyPar-Flow: 1) **Frontera** at Texas Advanced Computing Center (TACC), 2) **Stampede2** (Skylake partition) at TACC, and 3) **Epyc**: A local system with dual-socket AMD EPYC 7551 32-core processors.

Inter-connect: Frontera nodes are connected using Mellanox InfiniBand HDR-100 HCAs whereas Stampede2 nodes are connected using Intel Omni-Path HFIs.

DL Framework: All experiments have been performed using TensorFlow v1.13.

MPI Library: MVAPICH2 2.3.2 was used on Frontera, Intel MPI 2018 was used on Stampede2, and MVAPICH2 2.3.1 was used on Epyc.

Model Definitions: We use and modify model definitions for VGG and ResNet(s) presented in Keras Applications/Examples [1].

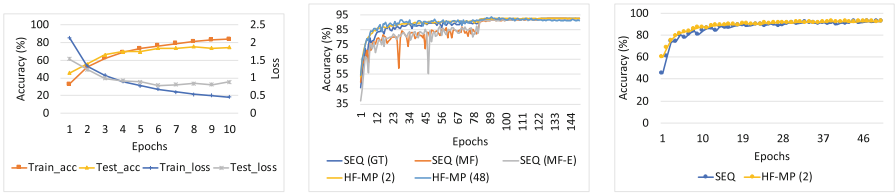
Note about GPUs: The design schemes proposed for HyPar-Flow are architecture-agnostic and can work on CPUs and/or GPUs. However, in this paper, we focus only on designs and scale-up/scale-out performance of many-core CPU clusters. We plan to perform in-depth GPU-based HyPar-Flow studies in the future.

We now present correctness related experiments followed by a comprehensive performance evaluation section.

6.1 Verifying the Correctness of HyPar-Flow

Because we propose and design HyPar-Flow as a new system, it is important to provide confidence to the users that HyPar-Flow not only offers excellent performance but also trains the model correctly. To this end, we present the correctness results based on two types of accuracy-related metrics: 1) Train Accuracy (train_acc)- Percentage of correct predictions for the training data during the training process and 2) Test Accuracy (test_acc)- Percentage of correct predictions for the testing data on the trained model. Both metrics are covered for small scale training using **VGG-16** on the CIFAR-10 dataset. We train VGG-16 for 10 epochs using 8 model-partitions on two Stampede2 nodes with a batch size of 128 and 16 pipeline stages as shown in Fig. 6(a). Next, we show test accuracy for **ResNet-110-v1** in Fig. 6(b) and **ResNet-1001-v2** in Fig. 6(c). The learning rate (LR) schedule was used from Keras Applications [1] for both ResNet(s) and was kept similar for sequential as well as parallel training variants. Training for ResNet-110 and ResNet-1001 was performed for 150 and 50 epochs, respectively. The following variants have been compared:

1) **SEQ (GT)** - Sequential using tf.GradientTape (GT).
2) **SEQ (MF)** - Sequential using model.fit (MF).
3) **SEQ (MF-E)** - Sequential using model.fit (MF) and (E)ager Execution.
4) **HF-MP (2)/(56)** - HyPar-Flow model-parallel with 2/56 model-partitions.



(a) VGG-16 Training (all metrics) for 10 epochs with BS=128 and LR=0.0002

(b) ResNet-110-v1 Test Accuracy for 150 Epochs with BS=32

(c) ResNet-1001-v2 Test Accuracy for 50 epochs with BS=32

Fig. 6. Testing the correctness of HyPar-Flow using different models

Discussion: Clearly, model-parallel training with HyPar-Flow is meeting the accuracy of the sequential model for 150 and 50 epochs of training for ResNet-110 and ResNet-1001, respectively. We note that training is a stochastic process and there are variations in earlier epochs whether we use the sequential version or the model-parallel version. However, the significance is of the end result, which in this case peaks at 92.5% for all the configurations presented. We ran multiple training jobs to ensure that the trends presented are reproducible.

6.2 Experimental Setup for Performance Evaluation

We use the term "process" to refer to a single *MPI Process* in this section. The actual mapping of the process to the compute units (or cores) varies according to the parallelization strategy being used. *Images/second* (or Img/sec) is the *metric* we are using for performance evaluation of different types of training experiments. Number of images processed by the DNN during training is affected by the *depth* (number of layers) of the model, batch size (*bs*), image size (W × H), and number of processes. Higher *Img/sec* indicates better performance. Some important terms are clarified further:

Batch Size (BS): # of samples in the batch (mini-batch)
Effective Batch Size (EBS) = BS × num_replicas for data/hybrid parallelism
Effective Batch Size (EBS) = BS for model-parallelism
Image Size: Dimension of the image (Width × Height).

Legend Entries for Graphs in Sects. 6.3 and 6.4 are:

- **Sequential:** Single-process DNN training using default TF/Keras APIs.
- **HF (MP):** DNN training using *hf.fit (..,strategy=model-parallel).*
- **HF (DP):** DNN training using *hf.fit(..,strategy=data-parallel).*
- **Horovod (DP):** DNN training using Horovod directly (data-parallel).

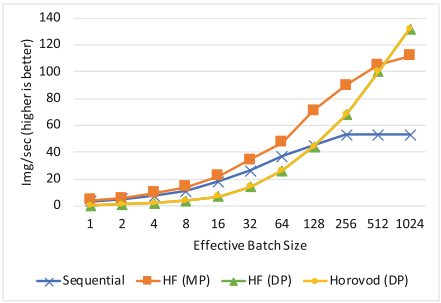
6.3 Model Parallelism on a Single Node

We train various models on a single Stampede2 node– dual-socket Xeon Skylake with 48 cores and 96 threads (hyper-threading enabled). The default version of TensorFlow relies on underlying math libraries like OpenBLAS and Intel MKL. On Intel systems, we tried the Intel-optimized version of TensorFlow, but it failed with different errors such as "function not implemented" etc. For the AMD system, we used the OpenBLAS available on the system. Both of these platforms offer very slow sequential training. *We present single-node results for VGG-16, ResNet-110-v1, and ResNet-1001-v2.*

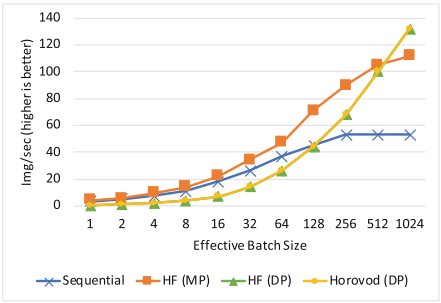
VGG-16 has 16 layers so it can be split in to as many as 16 partitions. We try all possible cases and observe the best performance for num_partitions = 8. As shown in Fig. 7(a), we see that HF (MP) offers better performance for small batch sizes and HF/Horovod (DP) offers better performance for large batch sizes. HF (MP) offers better performance compared to sequential (1.65× better at BS 1024) as well as to data-parallel training (1.25× better at BS 64) for VGG-16 on Stampede2.

ResNet-110-v1 has 110 layers so we were able to exploit up to 48 model-partitions within the node as shown in Fig. 7(b). We observe the following: 1) HF (MP) is up to 2.1× better than sequential at BS = 1024, 2) HF (MP) is up to 1.6× better than Horovod (DP) and HF (DP) at BS = 128, and 3) HF (MP) is 15% slower than HF (DP) at BS = 1024. The results highlight that model-parallelism is better at smaller batch sizes and data-parallelism are better only when large batch-size is used. Figure 8(a) shows that HF (MP) can offer up to

3.2× better performance than sequential training for ResNet-110-v1 on *Epyc* (64 cores). *Epyc* offered better scalability with increasing batch sizes compared to Stampede2 nodes (Fig. 7(b) vs. 8(a)) The performance gains suggest that HF (MP) can better utilize all cores on *Eypc* compared to sequential training.

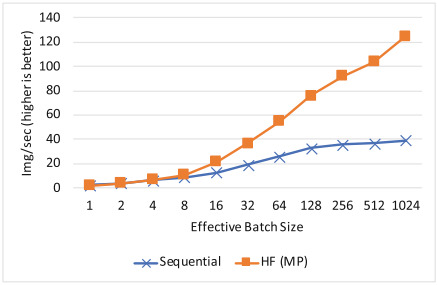


(a) VGG-16 up to 8 model-partitions

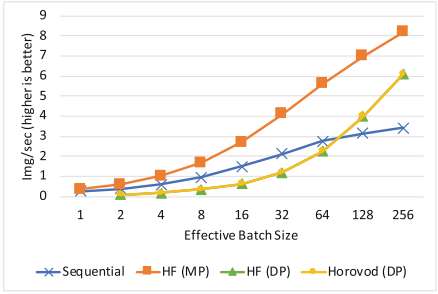


(b) ResNet-110-v1 up to 48 model-partitions

Fig. 7. HyPar-Flow's model-parallelism vs. sequential/data-parallelism (one node)



(a) ResNet-110-v1 up to 64 model-partitions



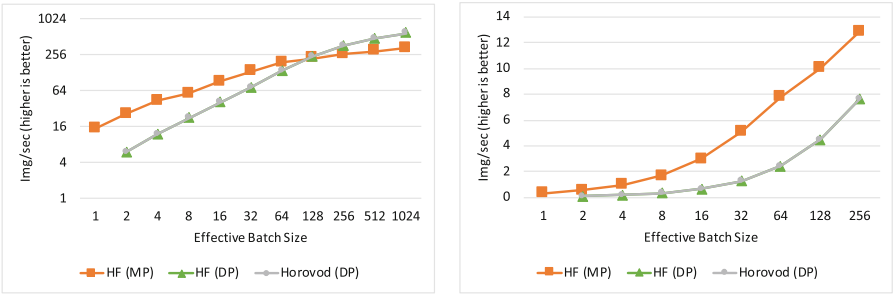
(b) ResNet-1001-v2 up to 48 model-partitions

Fig. 8. HyPar-Flow's model-parallelism vs. sequential/data-parallelism (one node)

ResNet-1001-v2: To push the envelope of model depth and stress the proposed HyPar-Flow system, we also perform experiments for ResNet-1001-v2, which has 1,0001 layers and approximately 30 million parameters. Figure 8(b) shows the performance for ResNet-1001-v2. It is interesting to note that data-parallel training performs poorly for this model. This is because the number of parameters increases the synchronization overhead for HF (DP) and Horovod (DP) significantly. Hence, even for large batch sizes, the computation is not enough to amortize the communication overhead. Thus, HF (MP) offers much better performance compared to sequential (2.4× better at BS = 256) as well as to data-parallel training (1.75× better at BS = 128).

6.4 Model Parallelism on Two Nodes

Two-node results for model parallelism are presented using VGG-16 and ResNet-1001-v2. Figure 9(a) shows the performance trends for VGG-16 training across two nodes. As mentioned earlier, we are only able to achieve good performance with model-parallelism for up to 8 model-partitions for the 16 layers of VGG-16. We also perform experiments for 16 model-partitions but observe performance degradation. This is expected because of the lesser computation per partition and greater communication overhead in this scenario. We scale ResNet-1001-v2 on two nodes using 96 model-partitions in the model-parallelism-only configuration on Stampede2. The result is presented in Fig. 9(b). We observe that model-parallel HF (MP) training provides 1.6× speedup (at BS = 256) over HF (DP) and Horovod (DP). On the other hand, a data-parallel-only configuration is not able to achieve good performance for ResNet-1001 due to significant communication (allreduce) overhead during gradient aggregation.



(a) VGG-16: MP Good for Small BS vs. DP Good for Large BS (8 model-partitions)

(b) ResNet-1001-v2: MP Good for All BS (up to 96 model-partitions).

Fig. 9. HyPar-Flow model-parallelism on two nodes

6.5 Hybrid Parallelism on Two Nodes (AmoebaNet)

Emerging models like AmoebaNet [19] are different compared to VGG and ResNet(s). In order to show the benefit of HyPar-Flow as a generic system for various types of models, we show the performance of training a 1,381-layer AmoebaNet variant in Fig. 10. We provide results for four different conditions: 1) Sequential training using Keras and TensorFlow on one node, 2) HF (MP) with 4 partitions on one node, 3) HF (MP) with 8 partitions on two nodes, and 4) HF (HP), where HP denotes hybrid parallelism on two nodes. As shown in Fig. 10, we observe that hybrid parallelism offers the best possible performance using the same set of nodes.

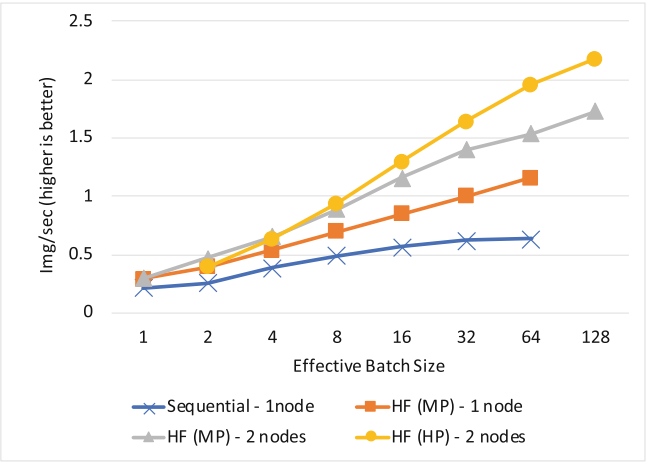
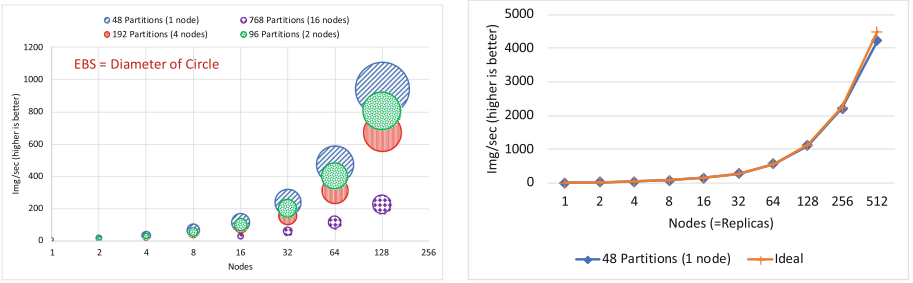


Fig. 10. Hybrid parallelism for AmoebaNet on two nodes

6.6 Hybrid Parallelism at Scale: Up to 28,762 Cores on 512 Nodes

The most comprehensive coverage of HyPar-Flow's flexibility, performance, and scalability are presented in Fig. 11(a). The figure shows performance for various combinations of hybrid-parallel training of ResNet-1001-v2 on 128 Stampede2 nodes. The figure has three dimensions: 1) the number of nodes on the X-axis, 2) Performance (Img/sec) on Y-axis, and 3) Batch Size using the diameter of the circles. The key takeaway is that hybrid-parallelism offers the user to make trade-offs between high-throughput (Img/sec) and batch size. From an accuracy (convergence) standpoint, the goal is to keep the batch-size small so model updates are more frequent. However, larger batch-size delays synchronization and thus provides higher throughput (Img/sec). HyPar-Flow offers the flexibility to control these two goals via different configurations. For instance, the large blue circle with diagonal lines shows results for 128 nodes using 128 model-replicas where the model is split into 48 partitions on the single 48-core node. This leads to a batch-size of just 32,768, which is 2× smaller than the expected 65,536 if pure data-parallelism is used. It is worth noting that the performance of pure data-parallelism even with 2× larger batch-size will still be lesser than the hybrid-parallel case, i.e., 793 img/sec (=6.2 × 128 – considering ideal scaling for data-parallel case presented earlier in Fig. 8(b)) vs. 940 img/sec (observed value– Fig. 11(a)). This is a significant benefit of hybrid-parallel training, which is impossible with pure model and/or data parallelism. In addition to this, we also present the largest scale we know of for any model/hybrid-parallel study on the latest Frontera system. Figure 11(b)) shows near-ideal scaling on 512 Frontera nodes. Effectively, every single core out of the 28,762 cores on these 512 nodes is being utilized by HyPar-Flow. The ResNet-1001 model is split into 56 partitions as Frontera nodes have a dual-socket Cascade-Lake Xeon processor for a total of 56 cores/node. We run one model-replica per node with a batch

size of 128. To get the best performance, pipeline stages were tuned and the best number was found to be 128.



(a) 128 Stampede2 nodes (b) 512 Frontera nodes

Fig. 11. Hybrid-parallelism at scale: ResNet-1001-v2 on Stampede and Frontera with different batch sizes, number of replicas, and number of partitions

6.7 Next-Generation Models: ResNet-5000?

Today, designers develop models accounting for the restriction of memory consumption. However, with HyPar-Flow, this restriction no longer exists, and designers can come up with models with as many layers as needed to achieve the desired accuracy. To illustrate this, we present ResNet-5000, an experimental model with 5000 layers. ResNet-5000 is massive and requires a lot of memory so we were able to train it with a batch-size of 1 only. Beyond that, it is not trainable on any existing system. We stress-test HyPar-Flow to scale the training of ResNet-5000 to two nodes and were able to train for bigger batch sizes. We note that training ResNet-5000 and investigation of its accuracy and finding the right set of hyper-parameters is beyond the scope of this paper. The objective is to showcase HyPar-Flow's ability to deal with models that do not exist today.

6.8 Discussion and Summary of Results

Model and data-parallelism can be combined in a myriad of ways to realize hybrid-parallel training. E.g. model-parallelism on a single node with multiple cores with data-parallelism across nodes. There are non-trivial and model-dependent trade-offs involved when designing hybrid schemes. Model-parallelism and data-parallelism have different use cases; model-parallelism is beneficial when we have a large model, or we want to keep a small effective batch size for training. On the other hand, data-parallelism gives a near-linear scale-out on multiple nodes but it also increases batch size. In our experiments, we observe that single-node model-parallelism is better than single-node data-parallelism. Theoretically, the number of model-partitions can not be larger than the number of layers in the model; we can not have more than 110 partitions for ResNet-110. In practice, however, we observe that one layer per model-partition will not

be used because it suffers from performance degradation. To conclude, HyPar-Flow's flexible hybrid-parallelism offers the best of both worlds; we can benefit from both model and data parallelism for the same model. We summarize the key observations below:

- Models like ResNet-110 offer better performance for model-parallelism on smaller batch sizes (<128).
- Newer and very-deep models like ResNet-1001 benefit from model-parallelism for any batch size (Fig. 8(b)).
- HyPar-Flow's model-parallel training provides up to 3.2× speedup over sequential training and 1.6× speedup over data-parallel training (Fig. 8(a)).
- HyPar-Flow's hybrid-parallel training offers flexible configurations and provides excellent performance for ResNet-1001; 110× speedup over single-node training on 128 Stampede2 (Xeon Skylake) nodes (Fig. 11(a)).
- HyPar-Flow's hybrid-parallel training is highly scalable; we scale ResNet-1001 to 512 Frontera nodes (28,762 cores) as shown in Fig. 11(b).

7 Conclusion

Deep Learning workloads are going through a rapid change as newer models and larger, more diverse datasets are being developed. This has led to an explosion of software frameworks like TensorFlow and approaches like data and model-parallelism to deal with ever-increasing workloads. In this paper, we explored a new approach to train state-of-the-art DNNs and presented HyPar-Flow: a unified framework that enables user-transparent and parallel training of TensorFlow models using multiple parallelization strategies. HyPar-Flow does not enforce any specific paradigm. It allows the programmers to experiment with different parallelization strategies without requiring any changes to the model definition and without the need for any system-specific parallel training code. Instead, HyPar-Flow Trainer and Communication Engine take care of assigning the partitions to different processes and performing inter-partition and inter-replica communication efficiently. For ResNet-1001 training using HyPar-Flow, we were able to achieve excellent speedups: up to 1.6× over data-parallel training, up to 110× over single-node training on 128 Stampede2 nodes, and up to 481× over single-node on 512 Frontera nodes. We also tested the ability of HyPar-Flow to train very large experimental models like ResNet-5000, which consists of 5,000 layers. We believe that this study paves new ways to design models. We plan to publicly release the HyPar-Flow system so that the community can use it to develop and train next-generation models on large-scale HPC systems.

References

1. Keras (2019). https://keras.io/
2. Model parallelism in MXNet (2019). https://mxnet.apache.org/api/faq/model_parallel_lstm

3. Akiba, T., Suzuki, S., Fukuda, K.: Extremely large minibatch SGD: training resnet-50 on ImageNet in 15 minutes (2017). CoRR abs/1711.04325. http://arxiv.org/abs/1711.04325
4. Awan, A.A., Chu, C., Subramoni, H., Lu, X., Panda, D.K.: OC-DNN: exploiting advanced unified memory capabilities in CUDA 9 and volta GPUs for out-of-core DNN training. In: 2018 IEEE 25th International Conference on High Performance Computing (HiPC), pp. 143–152, December 2018. https://doi.org/10.1109/HiPC.2018.00024
5. Awan, A.A., Hamidouche, K., Hashmi, J.M., Panda, D.K.: S-Caffe: co-designing MPI runtimes and caffe for scalable deep learning on modern GPU Clusters. In: Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPoPP 2017, pp. 193–205. ACM, New York (2017). https://doi.org/10.1145/3018743.3018769
6. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: expressing locality and independence with logical regions. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis SC 2012, pp. 66:1–66:11. IEEE Computer Society Press, Los Alamitos (2012). http://dl.acm.org/citation.cfm?id=2388996.2389086
7. Ben-Nun, T., Hoefler, T.: Demystifying parallel and distributed deep learning: an in-depth concurrency analysis (2018). CoRR abs/1802.09941. http://arxiv.org/abs/1802.09941
8. Dryden, N., Maruyama, N., Benson, T., Moon, T., Snir, M., Essen, B.V.: Improving strong-scaling of CNN training by exploiting finer-grained parallelism (2019). CoRR abs/1903.06681. http://arxiv.org/abs/1903.06681
9. Gholami, A., Azad, A., Jin, P., Keutzer, K., Buluc, A.: Integrated model, batch, and domain parallelism in training neural networks. In: Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures SPAA 2018, pp. 77–86. ACM, New York (2018). https://doi.org/10.1145/3210377.3210394
10. Goyal, P., et al.: Accurate, large minibatch SGD: training ImageNet in 1 hour (2017). CoRR abs/1706.02677. http://arxiv.org/abs/1706.02677
11. Harlap, A., et al.: PipeDream: fast and efficient pipeline parallel DNN training (2018). CoRR abs/1806.03377. http://arxiv.org/abs/1806.03377
12. He, K., Zhang, X., Ren, S., Sun, J.: Identity mappings in deep residual networks (2016). CoRR absscaffe,/1603.05027. http://arxiv.org/abs/1603.05027
13. Huang, Y., et al.: GPipe: efficient training of giant neural networks using pipeline parallelism (2018). CoRR abs/1811.06965. http://arxiv.org/abs/1811.06965
14. Jia, Z., Zaharia, M., Aiken, A.: Beyond data and model parallelism for deep neural networks (2018). CoRR abs/1807.05358. http://arxiv.org/abs/1807.05358
15. Krizhevsky, A.: One weird trick for parallelizing convolutional neural networks (2014). CoRR abs/1404.5997. http://arxiv.org/abs/1404.5997
16. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. In: Pereira, F., Burges, C.J.C., Bottou, L., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems, vol. 25, pp. 1097–1105. Curran Associates, Inc. (2012). http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf
17. Markthub, P., Belviranli, M.E., Lee, S., Vetter, J.S., Matsuoka, S.: DRAGON: breaking GPU memory capacity limits with direct NVM access. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis SC 2018, pp. 32:1–32:13. IEEE Press, Piscataway (2018). http://dl.acm.org/citation.cfm?id=3291656.3291699

18. Mikami, H., Suganuma, H., Chupala, U.-P., Tanaka, Y., Kageyama, Y.: Imagenet/resnet-50 training in 224 seconds (2018). CoRR abs/1811.05233. http://arxiv.org/abs/1811.05233
19. Real, E., Aggarwal, A., Huang, Y., Le, Q.V.: Regularized evolution for image classifier architecture search (2018). CoRR abs/1802.01548. http://arxiv.org/abs/1802.01548
20. Shazeer et al.: Mesh-TensorFlow: deep learning for supercomputers. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems, vol. 31, pp. 10414–10423. Curran Associates, Inc. (2018). http://papers.nips.cc/paper/8242-mesh-tensorflow-deep-learning-for-supercomputers.pdf
21. Sun, P., Feng, W., Han, R., Yan, S., Wen, Y.: Optimizing network performance for distributed DNN training on GPU clusters: Imagenet/alexnet training in 1.5 minutes (2019). CoRR abs/1902.06855. http://arxiv.org/abs/1902.06855