Towards a Scalable and Distributed Infrastructure for Deep Learning Applications

A Preprint

Bita Hasheminezhad, Shahrzad Shirzad, Nanmiao Wu, Patrick Diehl, Hartmut Kaiser[*]
Center of Computation & Technology
Louisiana State University, Baton Rouge, LA, U.S.A.
{bhashe1,sshirz1,wnanmi1}@lsu.edu
{pdiehl,hkaiser}@cct.lsu.edu

Hannes Schulz
Microsoft Research Montréal
Montréal, QC, Canada
hannes.schulz@microsoft.com

arXiv:2010.03012v1 [cs.DC] 6 Oct 2020

October 8, 2020

Abstract

Although recent scaling up approaches to train deep neural networks have proven to be effective, the computational intensity of large and complex models, as well as the availability of large-scale datasets require deep learning frameworks to utilize scaling out techniques. Parallelization approaches and distribution requirements are not considered in the primary designs of most available distributed deep learning frameworks and most of them still are not able to perform effective and efficient fine-grained inter-node communication. We present Phylanx that has the potential to alleviate these shortcomings. Phylanx presents a productivity-oriented frontend where user Python code is translated to a futurized execution tree that can be executed efficiently on multiple nodes using the C++ standard library for parallelism and concurrency (HPX), leveraging fine-grained threading and an active messaging task-based runtime system.

***K*eywords** Distributed Deep Learning · High Performance Computing · HPX · Asynchronous Manytask System

1 Introduction

The recent availability of large-scale datasets and ample computational power has triggered advances in a wide range of Deep Learning (DL) applications. Training a Deep Neural Network (DNN) is an iterative time-consuming process. The slope of advances decreases unless a reasonable training time is maintained. As such, a DL framework must be capable of training the DNN on multiple nodes (be distributed) and efficiently utilize resources on each node.

Most existing DL frameworks are not primarily designed to support major parallelization approaches or work in distributed. In many cases, training a DNN is not investigated as a High Performance Computing (HPC) problem; small models that fit in a single node are developed then scaled as needed. Thus, considerable efforts are required to make those DL frameworks compatible with the requirements of efficient scaling out, namely a distributed address space.

Using HPC capabilities to train DNNs is extremely advantageous. To efficiently utilize resources, the DL framework must support overlapping of communication and computation. There are HPC frameworks that are designed to overlap communication with useful computation adopting fine-grained parallelism. Message-driven communication, adaptive grain size optimization, and moving work to data are other HPC techniques that can be employed to improve the performance in DL arena [1, 2, 3].

[*]The STE||AR Group - https://stellar-group.org

In this paper, we make the following contributions: *1)* we revisit the requirements for scaling the training of DNNs (section 2), *2)* we evaluate the existing distributed DL frameworks based on these requirements (Section 3), and finally *3)* we introduce Phylanx as a distributed framework for DL applications. Finally, and we discuss some of Phylanx's design aspects and show primary Scaling Results (Section 4).

2 Requirements for Scalable Training of DNNs

There is a consensus that DL frameworks must strive to excel at two fundamental classes of characteristics: scalability and ease of use. Many frameworks attempt to put performance and usability at the center of their design [4, 5]. Some suggest that fine-grained parallelism is the key to achieve high performance [6, 7]. In this section, we elaborate on the requirements for such a DL framework.

2.1 Scale up and Scale out

To acquire a reasonable execution time for training modern models on large-scale datasets, it is necessary to exploit every chance of scaling up (scaling on a single node) and scaling out (scaling on multiple nodes) the training process. A single node, even the most powerful one, can never satisfy the memory requirements of contemporary DNNs. In the following, we go through common parallelization techniques usually applied to divide work across workers (compute resources). After that, we lay out the principles that enable as much resource utilization as possible to improve the parallel efficiency of the training process, and thus decrease the overall time required to train a DNN.

Deep Learning Parallelization Approaches

There are three major approaches for DNN parallelization; Data Parallelism, Model Parallelism, and Pipelining. Hybrid Parallelism is any combination of the aforementioned approaches.

Data Parallelism maintains an identical copy of the entire DNN model in addition to a partition of samples within a minibatch of data on each worker. DNN training is comprised of 3 stages: the *forward pass*, to produce activations and loss using current parameters and labels for the given samples, the *backward pass*, to compute gradients using the loss and parameters generated by the forward pass, and the *solver*, to determine the update rules to update the parameters. In the data parallelism approach, every computational resource independently computes the first two stages. The solver uses a collective communication operation for updating the parameters of the network at the end of each minibatch of data to keep model replicas consistent. Data parallelism is the most prevalent approach among DL frameworks to scale out the DNN training.

Model Parallelism splits the DNN along an iteration into disjoint or overlapped subsets and trains each subset on one worker. Model parallelism does not have a unanimous definition among the DL community; some define it as splitting the DNN along model parameters wherein a computational resource has only a portion of model parameters [7], some specify it as splitting channels or spatial dimensions [8]. In this paper, we define model parallelism as intra-iteration parallelism on any of the tensor dimensions except for the sample dimension. With this broad definition, spatial parallelism is a subset of model parallelism even though it requires all parameters on every computational resource. Model parallelism might incur extra communication. For example, disjoint spatial parallelism requires halo exchange in the forward and backward passes [9]. Model parallelism is crucial in training novel, wide and deep models. As effectively splitting the DNNs is non-trivial, model parallelism might impose considerable communication overhead if not designed properly.

Pipelining is a cross-iteration parallelism approach that assigns each consecutive set of layers in the model to a worker. In very deep neural networks, pipelining the work between workers might be beneficial. The data transfer between workers is comprised of activations and gradients. Pipelining is susceptible to the under-utilization of resources. In a naïve implementation of pipelining, only one worker is active at a time due to the sequential dependency of DNNs. The currently active worker performs one of the computationally intensive stages of training on a minibatch, *i.e.* forward or backward pass. The next worker has to wait to get the calculated activations or gradients from the current worker. Interleaving microbatch computations, and memorizing the result of calculations were proposed to parallelize this process [10, 11].

A pure data parallelization approach is not sufficient for DNN scaling. It is limited by minibatch size and model memory consumption. Although a larger minibatch size means a larger ratio of computation to communication, the minibatch size cannot grow arbitrarily. Memory constraints and accuracy degradation, which slows down convergence and diminishes generalization, [12, 13] impede this growth. Besides, the entire model should fit in the memory even for training on a single sample, which is impossible to achieve for large models. A large-scale DNN training requires hybrid parallelism to scale. This hybrid approach can be a combination of different approaches for different layers of

the network, *e.g.* data parallelism for layers with few parameters (mainly convolutional layers) and model parallelism for layers that are dense in parameters (fully-connected layers) [14]. A combination of different approaches on each of the layers of the network could be desired to satisfy the necessary granularity [9, 15]. Finally, a hybrid approach can use a distributed storage for model parameters, activations, and gradients at the expense of additional communication [16].

Optimal Resource Utilization

The scalability of a system can be viewed as its ability to efficiently utilize an increasing amount of resources while still satisfying its functionality requirements. In order to scale, a distributed implementation of a DNN training must support overlapping communication and computation, and still maintain the desired training accuracy. Although training a DNN on large datasets is a computation/communication intensive process similar to HPC applications [8], many implementations of parallel paradigms, especially data parallelization, in current DL frameworks have adopted inefficient global barriers imposing unnecessary overheads. Additionally, most existing frameworks are not optimized for CPU clusters, notwithstanding the fact that communication optimization and system resiliency among these are well-established. In this subsection, we present three specifications for a system with optimal resource utilization. The desired framework must use asynchronous collectives, a fine-grained execution platform, and its communication model must be integrated into its internal DL component implementation.

The desired scalable DNN framework must utilize asynchronous collectives instead of an asynchronous solver. Updates with Asynchronous Stochastic Gradient Descent (ASGD) were popularized by DistBelief [17] and its successor, TensorFlow [18] to mitigate the straggler effect, meaning computational resources must wait for the slowest machine to complete a phase of computation. The ASGD solver has a low statistical efficiency because parameters in each parameter server are updated by every worker of that server whenever the gradient is ready. As such, one worker might complete an iteration when other workers are updating for the next iteration. To avoid staleness of gradient updates, ASGD imposes a small learning rate that causes slow convergence and scaling issues [19, 20]. That could be the reason that TensorFlow only supports synchronous updates in its distributed data parallelism strategy, MultiWorkerMirroredStrategy[2] and its distributed pipelining strategy, GPipe [10]. Conversely, synchronous updates with asynchronous collectives would allow the workers to fill the idle-time with useful work.

To hide the latencies by using asynchronous collectives, there exist a few conditions: fine-grained but not too small units of work, short context switching times, and greatly reduced synchronization overheads. It is common knowledge in the DL community that a fine-grained execution flow has a higher chance of better scalability. Most DL frameworks support partitioning tensors in the batch dimension, some also support partitioning in other dimensions or a combination of operation dimensions. The desired scalable framework must support partitioning beyond the operation dimensions to be able to efficiently utilize the resources [21]. In practice, tensor partitioning has non-zero overhead and cannot be overused. Also, there is a high variation in gradient sizes, and some of them are too small, far from the size for optimal communication [22]. Thus, the desired framework, when possible, must combine the small tensors together before performing the collective operation [23]. Adopting Asynchronous Many-Task (AMT) runtime systems [24] is a plausible solution to provide fine-grained parallelism beyond hybrid parallelization. Potential AMTs providing distributed computing capabilities are Uintah [25], Chapel [26], Charm++ [27], Kokkos [28], Legion [29], PaRSEC [30], and HPX [31] which are compared in [32].

To exploit every opportunity for optimization, the desired scalable DNN framework must be a unified system having communication integrated into internal implementations of DL components. The desired distributed DL framework must contain the functionality to efficiently read the data, preprocess it, and create the DNN model to train its parameters whilst employing communication between the nodes. It is highly inefficient to run the parts of this workload separately. For instance, ByteScheduler[2] is a generic communication scheduler that relies on a dependency proxy mechanism with a non-zero overhead to interact with the DL frameworks execution engines. ByteScheduler improves the performance by partitioning and rearranging tensor transmissions, but as it cannot modify the source code of the DL framework that it works on, it must use the DL framework Directed Acyclic Graph (DAG) to create a more refined DAG using different kinds of proxies. HyPar-Flow [15] is another example of a non-unified system discussed in section 3.1.3.

2.2 Easy to Use API

A desired scalable DL platform should have a simple interface that highly abstracts the DL components while being easy to debug. Python has become the de-facto language for the ML/DL community as open-source libraries like NumPy [33], SciPy [34], Pandas [35], and MatPlotLib [36] keep up with the high performance numerical analysis and visualization demand. Most DL frameworks have an available API in Python as it is coherent, simple, and readable. Not all of these DL frameworks succeed to present a highly abstract or easy to debug API, *e.g.* newer versions of TensorFlow

[2] `https://www.tensorflow.org/guide/distributed_training#multiworkermirroedstrategy`

have adopted Keras [37] as their default interface. Since debuggability is essential for novel models, the DL community is more receptive toward frameworks with imperative paradigms rather than declarative ones. As such, TensorFlow switched to eager execution as of v2.0.

Besides being intuitive and debuggable, a desired scalable DL framework should have a non-intrusive transition to utilize its parallel and distributed features without requiring the user to add architecture- or platform-specific code or configuration options. A user should not need to modify the DNN model to run it with or without accelerators and should not have to add extra code for setting the parameter servers. Still, the user must decide how many nodes and/or cores they want to use in data and/or model parallelization and/or pipelining, but they need not be aware of the cluster topology to train their DNNs. Therefore, being architecture- and platform-agnostic is an important characteristic of an easy to use API. Runtime-adaptive capabilities that enable automatic optimizations at runtime are an important feature, such as finding optimal grain-sizes during parallelization, or finding the best-possible data distributions depending on the evaluated expressions.

3 Current Distributed Deep Learning Frameworks

In this section, we scrutinize existing distributed DL frameworks. We focus on frameworks that are more recent and popular among users. We did not include pure communication schedulers [2, 23, 38] or frameworks that rely on new hardware [39, 40]. We evaluate these DL frameworks based on the requirements described in Section 2. We summarize our observations in Table 1 where the columns represent: Data Parallelism, Model Parallelism/Pipelining, Communication overlaps Computation, Sufficient Granularity, Unified, Easy to Use, Architecture Agnostic, License, and Reference. Note that *Sufficient Granularity* represents whether the DL framework can utilize sufficiently fine-grained parallelism. To adequately control the amount of work for the computations (grain size), the DL framework may utilize a combination of parallelization approaches or may use novel solutions. We believe parallelism can and should be applied to the granularity of an individual operation not just to whole layers [7].

3.1 TensorFlow

Although TensorFlow [41] has always natively supported distributed training, data parallelism, and model partitioning (as of v0.8), older versions of TensorFlow require the user to determine the placement of each operation on devices to run on multiple nodes. Communications scheduling is not supported in TensorFlow out-of-the-box. TensorFlow has a centralized architecture wherein the number of workers cannot grow arbitrarily [42].

TensorFlow also has extensions to support different parallelization approaches. As of v2.2, the Multi Worker Mirrored Strategy is introduced and integrated into TensorFlow for data parallelism. Its update rule is synchronous and it has communication and computation overlapped. Google, the developer of TensorFlow, has developed Mesh TensorFlow [43] and GPipe [10] on top of TensorFlow to support model parallelism and pipelining. Many other frameworks are built on top of TensorFlow. Here, we introduce HyPar-Flow, an implementation of hybrid parallelism for DNN training with a Keras-like interface.

3.1.1 Mesh TensorFlow

Mesh TensorFlow (MTF) is a language to lay down a general class of tensor computations that requires the cluster of processors to be identical and reliable to perform optimally [43]. Specifying the layout manually, parallelization is possible on any of the tensor dimensions. MTF considers data parallelism as splitting on the batch dimension and enables the user to experiment with any kind of intra-iteration parallelism. A graph of MTF compiles into a SPMD program that depends on MPI communications techniques such as all_reduce. High communication overheads are introduced by these all_reduce-like operations [10].

Utilizing MTF is not straightforward. The user must create the layout and take care of having a reasonable chunk size for each tensor based on the cluster topology (mesh of processors). Also, this layout design is restricted by MTF splitting rules, *e.g.* two dimensions of a tensor are not allowed to be split across the same mesh dimension. Some available features of MTF are only tested on accelerators and, in particular, Tensor Processing Units (TPUs). In some cases, although the GPUs are detected, most computations are run on CPUs. MTF is not compatible with the newer versions of TensorFlow (eager evaluations).

3.1.2 GPipe

GPipe is a pipeline parallelism library implemented under Lingvo [44] framework which is a TensorFlow framework focusing on sequence-to-sequence models. GPipe partitions operation in the forward and backward pass and allows

data transfer between neighboring partitions. Therefore, there is a lower bubble overhead in comparison to a naive pipeline parallelization. GPipe does not memorize the activations of layers. As such, it can scale for large models as the number of accelerators may increase immensely, however, it needs to re-compute the activations of layers for the backward pass. In other words, GPipe is a scalable solution that attempts to have a higher utilization in a pipeline, but a portion of this utilization belongs to the re-computation. GPipe can be combined with data parallelism for a better scale.

There are few examples of GPipe available. It is unknown if GPipe is compatible with the eager TensorFlow. GPipe works only on accelerators and it assumes a single layer fits in the memory of one accelerator.

3.1.3 HyPar-Flow

HyPar-Flow is an implementation of data, model, and hybrid (combination of data and model) parallelization on Eager TensorFlow using MPI for communication and Keras for interface[15]. HyPar-Flow uses Horovod (Section3.4) for pure data parallelism. It implements the hybrid parallelism by generating a representation of the Keras code in a distributed manner and an MPI communicator for each model partition to devise the communication and computation overlap. HyPar-Flow is a non-unified framework, and it recognizes TensorFlow as a separate unmodifiable framework. Since in TensorFlow there is no access to the gradients of other layers, HyPar-Flow has to add a layer-like structure before each TensorFlow's layer.

HyPar-Flow only requires the strategy, the number of model partitions, and the number of model replicas from the user to utilize them with every possible intra-iteration parallelization. Its debuggability depends on its backend, TensorFlow. HyPar-Flow is platform-agnostic and also optimized for many-core CPU clusters, too.

3.2 Caffe

Berkeley AI Research founded Convolutional Architecture for Fast Feature Embedding (Caffe) [45] DL framework which does not support distributed training out-of-the-box. Caffe is a define-and-run (declarative) framework that has three blocking phases for training a DNN. There are many extensions of Caffe that support distributed training centralized or decentralized; each focuses on a specific platform or communication library. FireCaffe [46] and MPI-Caffe [47] use MPI to respectively deploy data and model parallelism on multi-GPU clusters. FeCaffe [48] and Caffe Barista [49] are FPGA specializations of Caffe for Convolutional Neural Networks (CNNs). Intel-Caffe[3] supports data parallelism training on CPU-based clusters. NVIDIA®-Caffe[4] is not a distributed framework. Here we discuss S-Caffe [1] and NUMA-Caffe [3] further.

3.2.1 S-Caffe

S-Caffe or OSU-Caffe is a product of co-designing CUDA-Aware MPI runtime and Caffe for data parallelism on GPU clusters. To overlap three phases of training in Caffe, S-Caffe uses on-demand communication and multi-stage non-blocking collectives. Except for a helper thread in gradient aggregation, S-Caffe does not use multi-threading.

3.2.2 NUMA-Caffe

NUMA-Caffe is a distributed DL framework that compliments BVLC-Caffe and Intel-Caffe implementations. BLVC-Caffe and Intel-Caffe apply BLAS-level and batch-level parallelism, respectively. NUMA-Caffe adds two additional levels of parallelism: Non-Uniform Memory Access (NUMA) node-level parallelism and thread-level parallelism. It also eliminates many remote memory accesses providing automatic data localization. NUMA-Caffe is platform-specific and it works around Caffe's implementation and its focus is Convolutional Neural Networks (CNNs).

3.3 PyTorch DDP

PyTorch, a successor of Caffe2[5], is an imperative DL framework developed by Facebook using dynamic computation graphs and automatic differentiation [4]. PyTorch is easy to use, debug, and customize. However, it functions such that 10% of speed can be traded in order to acquire a considerably intuitive model. Some PyTorch designs are admittedly susceptible to certain corner cases. PyTorch Distributed Data Parallel [50] is an extra feature intercepted to PyTorch to perform DDP computations and is available as of v1.5. PyTorch RPC is developed to support model parallelism but, as of now, is a project in progress.

[3] `https://github.com/intel/caffe`
[4] `https://ngc.nvidia.com/catalog/containers/nvidia:caffe`
[5] `https://caffe2.ai`

PyTorch DDP utilizes some techniques that are engineered to increase performance based on practice. These techniques are gradient bucketing (adds a hyper-parameter, bucket, to launch each all_reduce. Small tensors bucket into one all_reduce operation), overlapping communication with computation (which depends on when the first bucket gets ready and the backward computation order), and skipping synchronization. As these techniques are tuned to practice, PyTorch admittedly asserts that some of the solutions are not perfect but are reliable approximations on with minimum overhead. These techniques and shortcomings are discussed in [50]. PyTorch mainly focuses on ease of use, and enables users with options in training their models. For instance, for trainings that require larger scales, developers can explore enabling no_sync mode (skipping synchronization) if it attains acceptable convergence speed.

3.4 Horovod

Horovod [5] is a stand-alone Python library for data parallelism using an optimized ring_allreduce collective and a tensor fusion algorithm that works on top of another DL framework. At first, Horovod only supported TensorFlow as the DL worker, but currently, it supports PyTorch and MXNET, too. Horovod completely replaces the parameter server-based optimizer of TensorFlow which underutilizes the resources because of its communication overhead [51] with its synchronous optimizer. It can almost achieve linear performance gain if the portion of parameters in the dense layers to all parameters is small, *e.g.* unlike VGG-16 [52], the dense layer of ResNet50 [53] has a small portion of all parameters. Horovod supports model partitioning but does not support model or pipeline parallelism, so it can train only models that fit into a single device (maybe with multiple GPUs). Although it has one of the most optimized asynchronous collectives, in the absence of granularity, the communication overhead significantly grows with the number of nodes [54].

Horovod has a simple API and its own profiling tool, Horovod Timeline. The transition to distributed on TensorFlow code is easier than the original TensorFlow code transition. It also has a Keras interface which is popular in the DL community. Horovod is optimized for GPUs but can work without GPUs as well.

3.5 FlexFlow

FlexFlow is a DL framework with an execution optimizer that can find a strategy for any intra-iteration hybrid parallelization [7]. The execution simulator is initialized with data parallelism as well as anther parallelization strategy that is selected at random. Unless it takes more than the time budget of searching, it can find the optimal strategy. While this strategy is the best for intra-iteration parallelization, communication between partitions is not necessarily optimal. FlexFlow is built on an AMT, the Legion [29] runtime and it is able to parallelize its task graph at the granularity of a single operation.

FlexFlow has a Keras-like and a native interface. Using FlexFlow's execution optimizer is rather simple; it only takes the cluster topology and the graph corresponding to the DNN. FlexFlow works only on GPUs, though currently many top clusters in the world are not equipped.

3.6 Chainer

Chainer is the pioneer DL framework with a Define-by-Run (imperative) paradigm [55]. PyTorch is inspired by Chainer but focuses more on performance; PyTorch DDP passes most of the intensive computations to its C++ core while Chainer is developed purely in Python. Chainer only supports data parallelism. It has a synchronous decentralized design that can be realized by all_reduce communication while communication scheduling is not supported.

Chainer is simple to use and debug, and this simplicity has been the focal point of its implementation. Basic knowledge of Python and neural networks is sufficient to use it. GPU runs are supported through CuPy and allow users to code in a CPU/GPU-agnostic environment [56]. ChainerCV is an add-on package specialized for computer vision tasks for prototyping ideas by non-experts.

3.7 CNTK

Computational Network ToolKit (CNTK) [57] is Microsoft's open-source declarative library for DL. CNTK has developed four algorithms for its solver: Data-Parallel SGD, Block Momentum SGD, Model Averaging SGD, and Data Parallel ASGD[6]. Data-Parallel SGD uses a trick for reducing the size of communication called 1-bit SGD which compresses the gradient values to a single bit. The difference between the original gradient and its quantized value is added to the next minibatch. Block Momentum SGD uses Blockwise Model Update and Filtering (BUMF) which requires resetting momentum to zero while maintaining frequent model synchronization to converge. Microsoft™ no

[6] https://docs.microsoft.com/en-us/cognitive-toolkit/multiple-gpus-and-machines

longer recommends Model Averaging SGD since it falls behind the Data-Parallel SGD and Block Momentum SGD. Data-Parallel ASGD can be useful for models that are less sensitive to noise. CNTK does not support model parallelism.

CNTK has APIs in Python, C#, C++, and BrainScript which is its domain specific language for defining networks. It provides a performance profiler in Python that generates a detailed profile log. Users can get more information to debug by plotting the underlying graph easily with the logging information[7]. CNTK has been one of the official Keras' backends as of v2.0. By changing the argument in the device setting, CNTK can easily switch between its CPUs and GPUs implementations.

3.8 BigDL

BigDL is a distributed DL framework for data parallelism on top of Spark [58]. It does not support model parallelism. Like TensorFlow, BigDL has a parameter server-style architecture, but unlike TensorFlow, it favors coarse-grained operations where data transformations are immutable. BigDL processes its calculations across the nodes through the InvokeAndWait function. On a CPU cluster, BigDL is faster in computations than TensorFlow, benefiting from its CPU optimizations, but it suffers from long communication delays due to its dependency on MapReduce framework [59].

BigDL is integrated into the Spark functional compute model and does not suffer from overheads due to the adaptation of frameworks. It has been developed by Intel™ Analytics to handle the stream of dynamic and messy data. TensorFlowOnSpark [60] and CaffeOnSpark [61] use a connector approach to connect to Spark, and both have an execution model that is very different from Spark. BigDL runs a series of a-couple-of-seconds Spark jobs which are scheduled by Spark. On large clusters, scheduling may become a bottleneck for Spark; therefore, on each server, BigDL is set to not launch more than one task with multiple threads in each iteration. A benefit of using Spark is that it is equipped with fault tolerance and a fair load balancing mechanism. BigDL is modeled after Torch[8] and is easy to use and debug. It works best on a single Intel™ Xeon CPU node.

3.9 SINGA

SINGA [62] is an imperative distributed DL framework and one of the few that is primarily designed considering scaling. SINGA supports different synchronous (Sandblaster and AllReduce) and asynchronous (Downpour and Distributed Hogwild) solvers and is able to run with different workers/servers configurations. It can have multiple worker/server groups running asynchronously while each group of workers runs synchronously; therefore, SINGA is not limited to the medium-sized clusters. SINGA is not well optimized, it assigns one thread to each processor. SINGA exhibits good scaling out results but lacks in scaling up and accuracy [63, 64]. It supports data and model parallelism.

Users of SINGA should have a clear understanding of its layer-based programming model. They can choose to run their program on multiple nodes, but they need to configure it first. SINGA has several built-in layers. Also, layers are customizable as long as they are consistent with TrainOnBatch algorithm. SINGA is one of the few frameworks which allows the user to manually partition the layer transparently using concatenation and slice layers; it is one of the aspects of its design for scalability. SINGA has APIs in C++ and Python and runs on clusters with or without accelerators.

3.10 MXNET-MPI

MXNET-MPI [65] is the extension of MXNET that replaces each worker in a parameter server architecture with a group of workers. Workers of each group are synced together using an MPI collective operation where the solver is synchronous SGD. Therefore, for data parallelism, MXNET-MPI performs better than a fully centralized parameter server dependent architecture or a fully decentralized architecture that synchronizes parameters in a blocking fashion. Thus, overall execution time is reduced, even though there is no explicit scheduling for overlapping communication and computation. Putting workers into groups and having both synchronous and asynchronous updates is similar to SINGA's design. MXNET does not support multi-node model parallelism.

The key-value store is the critical component for training the DNN with MXNET on multiple nodes. The v1.5.1 MXNET, does not support training on more than 4 GPUs and, the set-up for distributed training is not user-friendly [66].

3.11 DeepSpeed / ZeRO

ZeRO [67] approaches training large models by focusing on solving the memory limitation problem while attempting to minimize the overhead. To train models where the memory of a single GPU is the limiting factor, ZeRO partitions

[7] https://cntk.ai/pythondocs/Manual_How_to_debug.html
[8] http://torch.ch/

Table 1: Comparison between Distributed Deep Learning Frameworks

Framework	Data Par	Model Par/Pipeline	Comm overlaps Comp	Sufficient Granularity	Unified	Easy to Use	Archt. Agnostic	License	Reference
Mesh-TensorFlow	✓	✓				(-)		Apache-2.0	[43]
GPipe	compatible	✓	✓		✓	(-)		Apache-2.0	[10]
HyPar-Flow	✓	✓	✓	✓		(++)	✓	N/A	[15]
S-Caffe/OSU-Caffe	✓		✓		✓	N/A		N/A*	[1]
NUMA-Caffe	✓		✓			(+)		Public Domain	[3]
PyTorch DDP	✓		✓		✓	(++)	✓	BSD-style	[50]
Hororvod	✓	compatible	✓			(++)	✓	Apache-2.0	[5]
FlexFlow	✓	✓	✓	✓	✓	(+)		Apache-2.0	[50]
Chainer	✓				✓	(++)	✓	MIT	[55]
CNTK	✓		✓		✓	(++)	✓	MIT	[57]
BigDL	✓				✓	(+)		Apache-2.0	[58]
SINGA	✓	✓			✓	(+)	✓	Apache-2.0	[62]
MXNET-MPI	✓				✓	(++)		Apache-2.0	[65]
DeepSpeed	✓	compatible	✓	✓	✓	(++)		MIT	[16]
Phylanx	✓	✓	✓	✓	✓	(++)	✓	Boost-1.0	[68]

*Only available in binary form

Data Par shows if the framework supports Data Parallelism on multiple nodes. **Model Par/Pipeline** is checkmarked if the framework supports any network intra-iteration or inter-iteration parallelism other than data parallelism. **Comm overlaps Comp** highlights if there is any explicit attempt that prevents sequential run of computation and communication. **Sufficient Granularity** represents whether parallelism is applicable to the granularity of an individual operation. **Unified** is checkmarked if the infrastructure that makes the training distributed is integrated into the implementation of its DL components. **Easy to Use** has two measures: simple interface for coding and debugging. **Archt. Agnostic** means no modification is needed to use the code in distributed on different architectures or platforms. **License** and **Reference** are self-explanatory.

activations, optimizer states, gradients, and parameters and distributes them equally overall available nodes. It then employs overlapping collective operations to reconstruct the tensors as needed. This gives DeepSpeed the memory advantages of model parallelism and pipelining, while retaining the flexibility and ease of use of data parallelism.

The published DeepSpeed [16] implementation can be used as a drop-in replacement for PyTorch's DDP (Section 3.3) and can optimize any operation that is derived from the `pytorch.nn.module`. Due to this fine granularity, DeepSpeed can make tailored decisions about which tensors to distribute, whether to off-load memory to CPU and where to limit buffer sizes to prevent out-of-memory issues in the allocator. While the current release version does not include model parallelism or pipelining techniques, DeepSpeed is compatible with approaches that do.

4 Phylanx as a Deep Learning Framework

Phylanx [69, 68] is an asynchronous distributed array processing framework that combines the benefits of exposing a high-productivity Python-based development environment with a high-performance C++-based execution environment targeting computer systems of any size, including cloud and HPC-cluster technologies. The framework automatically transforms the user-provided Python code into an intermediate representation that is efficiently executed and distributed across all available compute resources as specified by the user.

4.1 How Phylanx tackles design challenges of DL frameworks

Phylanx is a software framework that is based on the HPX runtime system, which in turn was designed from first principles to address well known key challenges of high-performance computing applications. As such, Phylanx implicitly and naturally benefits from inheriting the advantages of applying fine-grained parallelism, message driven computation, constrained-based synchronization (never synchronize more than needed for the local progress of execution), implicit overlapping computation with communication, and runtime-adaptive granularity control. In this section we describe these capabilities on more detail and demonstrate that Phylanx addresses some of the identified challenges of DL frameworks (see Table 1). We are aware that Phylanx is no "jack of all trades" solution, but many of the challenges are tackled.

DL Paralleization approaches

In Phylanx, distribution of data is done by tiling the data arrays involved such that each locality the application runs on is responsible for one (or more) tiles of the data. Phylanx execution is strictly SPMD-style, *i.e.* each locality executes a structurally equivalent expression graph, while each part of that graph performs communication between the localities depending on the operations to be performed. The execution of this expression graph is done fully asynchronously, and the necessary communication relies on asynchronous collectives. This allows to fully overlap them with the ongoing computations. This approach allows to seamlessly distribute the processed data arrays across a possibly large amount of localities (*i.e.* nodes in a cluster) while maintaining the best possible scalability. Each of the tiles of the data arrays handled by a locality is internally represented exactly like a fully local data array with the exception that it

carries additional meta-information describing the whole (distributed) array. This has the benefit of simplifying the implementation.

Phylanx supports overlapped tiling which is beneficial in spatial parallelization. A halo exchange is needed in forward and backward pass using spatial parallelization. When the kernel size is comparably smaller than the data size, the user can use overlapped tiling to avoid extra communication.

Communication overlaps Computation

Phylanx uses the HPX runtime system as its underlying execution platform. HPX is the C++ Standard Library for Parallelism and Concurrency [70, 71, 31]. It provides the necessary abstractions to build efficient codes that are oblivious to local and remote operations while maintaining efficient data locality. HPX has been described in detail in other publications [72, 73, 74, 75, 76]. In the context of Phylanx, we use HPX because of its dynamic scheduling and global data addressing capabilities as well as its ISO C++ standards conforming API. The constructs it exposes are well-aligned with the existing C++ standards [77, 78, 79, 80]. HPX is fundamental for Phylanx, as it uses shared memory abstractions that have already been adopted in the most recent ISO C++ standards and HPX's distributed memory abstractions are standards conforming extensions. Using the *Futurization* technique in HPX, the execution of Phylanx code is expressed as complex dataflow execution graphs that can generate a large amount of fine-grained parallel tasks that are scheduled to execute only when their dependencies are satisfied (see for instance [81]). This naturally and intrinsically enables overlapping computation with communication, thus perfectly hiding communication latencies.

Phylanx achieves overlapping computation and communication using its asynchronous active messaging communication platform (that also includes its asynchronous collectives). Phylanx prefers moving work to data over moving data to work and to means of runtime-adaptively coalesce messages into larger units (tensor fusion) [82, 83], which further reduces the latencies and overheads caused by the necessary communication operations (see Section 4.1 for more information).

Sufficient Granularity

Most of the analyzed DL frameworks have mentioned fine-grained parallelism. The purpose is to have a synchronous optimizer while maintaining acceptable performance. However, with fine-grained parallelism, meaning many small computational tasks are executed, the overhead of context switching takes into account using the system threads.

Phylanx is utilizing HPX's light-weight user-level threading system to reduce the context switching and synchronization overheads of small computational tasks to its minimum [84, 85]. HPX's asynchronous execution model in combination with its active-message based communication model intrinsically allows to hide latencies exhibited by those communication operations, naturally overlaps those with useful computation, and reduces synchronization overheads across nodes to a minimum. HPX's asynchronous collectives allow to break the strict lock-stepping between ranks and enable them to perform other work while the collective operation is being performed.

In order to avoid for threads to become too short lived (which increases associated overheads), Phylanx employs runtime-adaptive techniques for controlling the granularity of tasks and the size of networking packages, both with the goal of reducing overheads and maximizing system utilization [82].

Unified

Phylanx exploits every chance of parallelism since it implements every required DL operation using HPX. Thus, the infrastructure that makes the training scale out is integrated into the DL framework. On the contrary, we observe that not considering scaling out requirements in designing a DL framework has left us with stitching different libraries (connector approach) or re-implementing components to achieve functionality.

Easy to Use

Phylanx provides a high-productivity debugable Python-based interactive interface, JetLag[9] [86]. JetLag constitutes a container-based Jupyter notebook interface, a performance visualization platform [87] (Traveler) and a performance measurement library (APEX [88]). The user can code in a Jupyter notebook and easily plot node-link diagram of the execution tree as well as Gantt and utilization charts using APEX performance counters [89].

[9] `https://github.com/STEllAR-GROUP/JetLag`

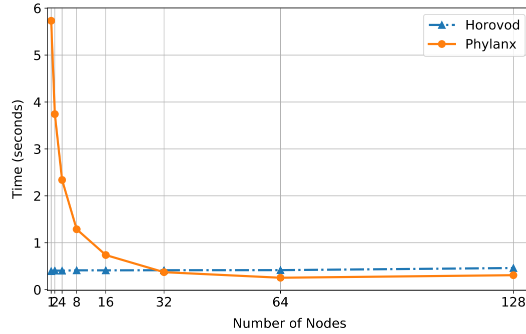

Figure 1: Comparison of Phylanx and Horovod executing forward propagation of a CNN on an HPC cluster up to 128 nodes.

Architecture Agnostic

Running Phylanx on shared memory or distributed memory is hidden from the user due to the high-level abstractions provided by HPX. A unified syntax and semantics for local and remote operations provided by the Adaptive Global Address Space (AGAS) [90, 91] is utilized. Thus, the user does not need to modify the Python code for local and remote computations or using another platform.

In addition to addressing the above challenges, Phylanx supports fault tolerance. Long-running non-dedicated resources can be used to train a large DNN; therefore, a means of fault tolerance is essential in the occurrence of a failure [18]. TensorFlow provides a checkpoint-based fault tolerance system while BigDL incorporates fault tolerance using Spark's RDD [92] on every operation. HPX provides software resilience [93] to implement fault tolerance within Phylanx as well. HPX can detect silent data corruptions, *e.g.* memory bit flips or CPU floating point errors. After some corrupted computation on a node, the user can do one of the following: *1)* replay the computation and use the new computation if the silent data corruption vanished. *2)* start replicates of the computation which are executed independently. There are three possibilities to compare the replicates: *a)* use check sums to compare; *b)* use a user-defined consensus function, which returns the replicate passing the tests; and *c)* if multiple replicates pass the consensus function, the user provides a validate function to decided which one to use. HPX aims to resolve the problems of scalability, resiliency, power efficiency, and runtime adaptive resource management that continue to grow in importance, as the industry is facing increasing demands in supporting highly distributed heterogeneous systems.

4.2 Primary Scaling Results

We have used an Intel™ Cascade Lake based cluster called Queen Bee 3 maintained by LONI [94]. This CPU cluster consists of 192 nodes each containing 2×24-Core Xeon Platinium 8260 processors. We have measured the execution time for the forward pass of a 4-layers CNN (deduced from Kaggle[10]) on a Human Activity Recognition data. For a mini-batch size of 8000, we compare the execution time of Phylanx (git hash: 03ad3b8) to Horovod v0.20.0, which is installed on top of TensorFlow v2.3.0 and uses Gloo (git hash: fe2ad9c) [95] as its communication library in Figure 1. First, it can be seen that the execution time of Horovod does not significantly decrease when using more nodes while that of Phylanx shows a notable reduce, which demonstrates the scalability of Phylanx. Second, we find that Phylanx takes a smaller execution time at least ($\approx 18\%$) in comparison to Horovod when using 32 or more nodes, which provides insight into running CNN on larger clusters.

5 Conclusion

With the ever-increasing need for reducing the time of training modern deep neural networks, scaling out has become the standard practice. To this effect, we revisited the requirements for distributed training of deep neural networks and described the underlying factors for the desired framework. It must provide *a)* asynchronous collectives, *b)* fine-grained execution platform, *c)* must be unified *d)* must be easy to use and debug, and *e)* must be architecture and platform agnostic. Most existing frameworks started as single-node computational models then adapted running on

[10] `https://www.kaggle.com/niyati11/project-convo1d`

GPU clusters. But we should design a system from first principles to embrace the challenges of platform-agnostic distributed computation. We showed that Phylanx offers great potential even in its current state.

Acknowledgements

The authors are grateful for the support of this work by the LSU Center for Computation & Technology and by the DTIC project: Phylanx Engine Enhancement and Visualizations Development (Contract Number: FA8075-14-D-0002/0007).

References

[1] Ammar Ahmad Awan, Khaled Hamidouche, Jahanzeb Maqbool Hashmi, and Dhabaleswar K Panda. 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*, pages 193–205, 2017.

[2] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 16–29, 2019.

[3] Probir Roy, Shuaiwen Leon Song, Sriram Krishnamoorthy, Abhinav Vishnu, Dipanjan Sengupta, and Xu Liu. Numa-caffe: Numa-aware deep learning neural networks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(2):1–26, 2018.

[4] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[5] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[6] Nikoli Dryden, Naoya Maruyama, Tim Moon, Tom Benson, Marc Snir, and Brian Van Essen. Channel and filter parallelism for large-scale cnn training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–20, 2019.

[7] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*, 2018.

[8] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

[9] Nikoli Dryden, Naoya Maruyama, Tom Benson, Tim Moon, Marc Snir, and Brian Van Essen. Improving strong-scaling of cnn training by exploiting finer-grained parallelism. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 210–220. IEEE, 2019.

[10] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in neural information processing systems*, pages 103–112, 2019.

[11] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

[12] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[13] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[14] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[15] Ammar Ahmad Awan, Arpan Jain, Quentin Anthony, Hari Subramoni, and Dhabaleswar K Panda. Hyparflow: Exploiting mpi and keras for scalable hybrid-parallel dnn training using tensorflow. *arXiv preprint arXiv:1911.05146*, 2019.

[16] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, page 3505–3506, 2020.

[17] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[18] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[19] Fan Zhou and Guojing Cong. On the convergence properties of a k-step averaging stochastic gradient descent algorithm for nonconvex optimization. *arXiv preprint arXiv:1708.01012*, 2017.

[20] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.

[21] Wenbin Jiang, Yangsong Zhang, Pai Liu, Jing Peng, Laurence T Yang, Geyan Ye, and Hai Jin. Exploiting potential of deep neural networks by layer-wise fine-grained parallelism. *Future Generation Computer Systems*, 102:210–221, 2020.

[22] Kevin Siu, Dylan Malone Stuart, Mostafa Mahmoud, and Andreas Moshovos. Memory requirements for convolutional neural network hardware accelerators. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 111–121. IEEE, 2018.

[23] Yixin Bao, Yanghua Peng, Yangrui Chen, and Chuan Wu. Preemptive all-reduce scheduling for expediting distributed dnn training. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 626–635. IEEE, 2020.

[24] Abhishek Kulkarni and Andrew Lumsdaine. A comparative study of asynchronous many-tasking runtimes: Cilk, charm++, parallex and am++. *arXiv preprint arXiv:1904.00518*, 2019.

[25] J Davison de St Germain, John McCorquodale, Steven G Parker, and Christopher R Johnson. Uintah: A massively parallel problem solving environment. In *Proceedings the Ninth International Symposium on High-Performance Distributed Computing*, pages 33–41. IEEE, 2000.

[26] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[27] Laxmikant V Kale and Sanjeev Krishnan. Charm++ a portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–108, 1993.

[28] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[29] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[30] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[31] Hartmut Kaiser, Patrick Diehl, Adrian S. Lemoine, Bryce Adelstein Lelbach, Parsa Amini, Agustín Berge, John Biddiscombe, Steven R. Brandt, Nikunj Gupta, Thomas Heller, Kevin Huck, Zahra Khatami, Alireza Kheirkhahan, Auriane Reverdell, Shahrzad Shirzad, Mikael Simberg, Bibek Wagle, Weile Wei, and Tianyi Zhang. Hpx - the c++ standard library for parallelism and concurrency. *Journal of Open Source Software*, 5(53):2352, 2020.

[32] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, et al. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing*, 74(4):1422–1434, 2018.

[33] Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in science & engineering*, 13(2):22–30, 2011.

[34] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020.

[35] Wes McKinney et al. pandas: a foundational python library for data analysis and statistics. *Python for High Performance and Scientific Computing*, 14(9), 2011.

[36] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(3):90–95, 2007.

[37] François Chollet et al. keras, 2015.

[38] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288*, 2018.

[39] Ali HeydariGorji, Mahdi Torabzadehkashi, Siavash Rezaei, Hossein Bobarshad, Vladimir Alves, and Pai H Chou. Stannis: Low-power acceleration of deep neuralnetwork training using computational storage. *arXiv preprint arXiv:2002.07215*, 2020.

[40] Ali HeydariGorji, Siavash Rezaei, Mahdi Torabzadehkashi, Hossein Bobarshad, Vladimir Alves, and Pai H Chou. Hypertune: Dynamic hyperparameter tuning for efficient distribution of dnn training over heterogeneous systems. *arXiv preprint arXiv:2007.08077*, 2020.

[41] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[42] Jingoo Han, Luna Xu, M Mustafa Rafique, Ali R Butt, and Seung-Hwan Lim. A quantitative study of deep learning training on heterogeneous supercomputers. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12. IEEE, 2019.

[43] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems*, pages 10414–10423, 2018.

[44] Jonathan Shen, Patrick Nguyen, Yonghui Wu, Zhifeng Chen, Mia X Chen, Ye Jia, Anjuli Kannan, Tara Sainath, Yuan Cao, Chung-Cheng Chiu, et al. Lingvo: a modular and scalable framework for sequence-to-sequence modeling. *arXiv preprint arXiv:1902.08295*, 2019.

[45] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.

[46] Forrest N Iandola, Matthew W Moskewicz, Khalid Ashraf, and Kurt Keutzer. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2592–2600, 2016.

[47] Stefan Lee, Senthil Purushwalkam, Michael Cogswell, David Crandall, and Dhruv Batra. Why m heads are better than one: Training a diverse ensemble of deep networks. *arXiv preprint arXiv:1511.06314*, 2015.

[48] Ke He, Bo Liu, Yu Zhang, Andrew Ling, and Dian Gu. Fecaffe: Fpga-enabled caffe with opencl for deep learning training and inference on intel stratix 10. *arXiv preprint arXiv:1911.08905*, 2019.

[49] Diederik Adriaan Vink, Aditya Rajagopal, Stylianos I Venieris, and Christos-Savvas Bouganis. Caffe barista: Brewing caffe with fpgas in the training loop. *arXiv preprint arXiv:2006.13829*, 2020.

[50] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[51] Ammar Ahmad Awan, Jeroen Bedorf, Ching-Hsiang Chu, Hari Subramoni, and Dhabaleswar K Panda. Scalable distributed dnn training using tensorflow and cuda-aware mpi: Characterization, designs, and performance evaluation. *arXiv preprint arXiv:1810.11112*, 2018.

[52] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[53] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[54] Xingfu Wu, Valerie Taylor, Justin M Wozniak, Rick Stevens, Thomas Brettin, and Fangfang Xia. Performance, power, and scalability analysis of the horovod implementation of the candle nt3 benchmark on the cray xc40 theta. In *SC18 Workshop on Python for High-Performance and Scientific Computing, Dallas, USA*, 2018.

[55] Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, Brian Vogel, and Hiroyuki Yamazaki Vincent. Chainer: A deep learning framework for accelerating the research cycle. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2002–2011, 2019.

[56] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, pages 1–6, 2015.

[57] Frank Seide and Amit Agarwal. Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2135–2135, 2016.

[58] Jason Jinquan Dai, Yiheng Wang, Xin Qiu, Ding Ding, Yao Zhang, Yanzhang Wang, Xianyan Jia, Cherry Li Zhang, Yan Wan, Zhichao Li, et al. Bigdl: A distributed deep learning framework for big data. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 50–60, 2019.

[59] Xin Du, Di Kuang, Yan Ye, Xinxin Li, Mengqiang Chen, Yunfei Du, and Weigang Wu. Comparative study of distributed deep learning tools on supercomputers. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 122–137. Springer, 2018.

[60] *TensorFlowOnSpark*, 2017.

[61] *CaffeOnSpark2017*, 2016.

[62] Beng Chin Ooi, Kian-Lee Tan, Sheng Wang, Wei Wang, Qingchao Cai, Gang Chen, Jinyang Gao, Zhaojing Luo, Anthony KH Tung, Yuan Wang, et al. Singa: A distributed deep learning platform. In *Proceedings of the 23rd ACM international conference on Multimedia*, pages 685–688, 2015.

[63] Shayan Shams, Richard Platania, Kisung Lee, and Seung-Jong Park. Evaluation of deep learning frameworks over different hpc architectures. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1389–1396. IEEE, 2017.

[64] Wei Wang, Gang Chen, Anh Tien Tuan Dinh, Jinyang Gao, Beng Chin Ooi, Kian-Lee Tan, and Sheng Wang. Singa: Putting deep learning in the hands of multimedia users. In *Proceedings of the 23rd ACM international conference on Multimedia*, pages 25–34, 2015.

[65] Amith R Mamidala, Georgios Kollias, Chris Ward, and Fausto Artico. Mxnet-mpi: Embedding mpi parallelism in parameter server task model for scaling deep learning. *arXiv preprint arXiv:1801.03855*, 2018.

[66] Sean Mahon, Sébastien Varrette, Valentin Plugaru, Frédéric Pinel, and Pascal Bouvry. Performance analysis of distributed and scalable deep learning. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 760–766. IEEE, 2020.

[67] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimization towards training a trillion parameter models. *arXiv preprint arXiv:1910.02054*, 2019.

[68] R. Tohid, B. Wagle, S. Shirzad, P. Diehl, A. Serio, A. Kheirkhahan, P. Amini, K. Williams, K. Isaacs, K. Huck, S. Brandt, and H. Kaiser. Asynchronous execution of python code on task-based runtime systems. In *2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, pages 37–45, 2018.

[69] Hartmut Kaiser and Parsa Amini and Steven R. Brandt and Bita Hasheminezhad and Bibek Wagle and R. Tohid and Nanmiao Wu and Shahrzad Shirzad. MPET - A C++ expression template library for arithmetic operations, 2020. https://github.com/STEllAR-GROUP/phylanx.

[70] Hartmut Kaiser, Bryce Adelstein Lelbach aka wash, Thomas Heller, Mikael Simberg, Agustín Bergé, John Biddiscombe, aurianer, Anton Bikineev, Grant Mercer, Andreas Schäfer, Kevin Huck, Adrian S. Lemoine, Taeguk Kwon, Jeroen Habraken, Matthew Anderson, Marcin Copik, Steven R. Brandt, Martin Stumpf, Daniel Bourgeois, Denis Blank, Shoshana Jakobovits, Vinay Amatya, rstobaugh, Lars Viklund, Zahra Khatami, Patrick Diehl, Tapasweni Pathak, Devang Bacharwar, Shuangyang Yang, and Erik Schnetter. STEllAR-GROUP/hpx: HPX V1.4.1: The C++ Standards Library for Parallelism and Concurrency, February 2020.

[71] Thomas Heller, Patrick Diehl, Zachary Byerly, John Biddiscombe, and Hartmut Kaiser. HPX – An open source C++ Standard Library for Parallelism and Concurrency. In *Proceedings of OpenSuCo 2017, Denver, Colorado USA, November 2017 (OpenSuCo 17)*, page 5, 2017.

[72] T. Heller, H. Kaiser, and K. Iglberger. Application of the ParalleX Execution Model to Stencil-based Problems. In *Proceedings of the International Supercomputing Conference ISC'12, Hamburg, Germany*, 2012.

[73] Thomas Heller, Hartmut Kaiser, Andreas Schäfer, and Dietmar Fey. Using HPX and LibGeoDecomp for Scaling HPC Applications on Heterogeneous Supercomputers. In *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '13, pages 1:1–1:8, New York, NY, USA, 2013. ACM.

[74] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. HPX: A Task Based Programming Model in a Global Address Space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA, 2014. ACM.

[75] Hartmut Kaiser, Thomas Heller, Daniel Bourgeois, and Dietmar Fey. Higher-level parallelization for local and distributed asynchronous task-based programming. In *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware*, ESPM '15, pages 29–37, New York, NY, USA, 2015. ACM.

[76] Thomas Heller, Hartmut Kaiser, Patrick Diehl, Dietmar Fey, and Marc Alexander Schweitzer. Closing the Performance Gap with Modern C++. In Michaela Taufer, Bernd Mohr, and Julian M. Kunkel, editors, *High Performance Computing*, volume 9945 of *Lecture Notes in Computer Science*, pages 18–31. Springer International Publishing, 2016.

[77] The C++ Standards Committee. ISO International Standard ISO/IEC 14882:2011, Programming Language C++. Technical report, Geneva, Switzerland: International Organization for Standardization (ISO)., 2011. http://www.open-std.org/jtc1/sc22/wg21.

[78] The C++ Standards Committee. ISO International Standard ISO/IEC 14882:2014, Programming Language C++. Technical report, Geneva, Switzerland: International Organization for Standardization (ISO)., 2014. http://www.open-std.org/jtc1/sc22/wg21.

[79] The C++ Standards Committee. ISO International Standard ISO/IEC 14882:2017, Programming Language C++. Technical report, Geneva, Switzerland: International Organization for Standardization (ISO)., 2017. http://www.open-std.org/jtc1/sc22/wg21.

[80] The C++ Standards Committee. ISO International Standard ISO/IEC 14882:2020, Programming Language C++. Technical report, Geneva, Switzerland: International Organization for Standardization (ISO)., 2020. http://www.open-std.org/jtc1/sc22/wg21.

[81] Thomas Heller, Bryce Adelstein Lelbach, Kevin A Huck, John Biddiscombe, Patricia Grubel, Alice E Koniges, Matthias Kretz, Dominic Marcello, David Pfander, Adrian Serio, Juhan Frank, Geoffrey C Clayton, Dirk Pflüger, David Eder, and Hartmut Kaiser. Harnessing billions of tasks for a scalable portable hydrodynamic simulation of the merger of two stars. *The International Journal of High Performance Computing Applications*, 0(0):1094342018819744, 2019.

[82] Bibek Wagle, Mohammad Alaul Haque Monil, Kevin Huck, Allen D Malony, Adrian Serio, and Hartmut Kaiser. Runtime adaptive task inlining on asynchronous multitasking runtime systems. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019.

[83] Bibek Wagle, Samuel Kellar, Adrian Serio, and Hartmut Kaiser. Methodology for adaptive active message coalescing in task based runtime systems. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1133–1140. IEEE, 2018.

[84] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. Parallex an advanced parallel execution model for scaling-impaired applications. In *2009 International Conference on Parallel Processing Workshops*, pages 394–401. IEEE, 2009.

[85] Gabriel Laberge, Shahrzad Shirzad, Patrick Diehl, Hartmut Kaiser, Serge Prudhomme, and A Lemoine. Scheduling optimization of parallel linear algebra algorithms using supervised learning. *arXiv preprint arXiv:1909.03947*, 2019.

[86] Steven R Brandt, Alex Bigelow, Sayef Azad Sakin, Katy Williams, Katherine E Isaacs, Kevin Huck, Rod Tohid, Bibek Wagle, Shahrzad Shirzad, and Hartmut Kaiser. Jetlag: An interactive, asynchronous array computing environment. In *Practice and Experience in Advanced Research Computing*, pages 8–12. 2020.

[87] Katy Williams, Alex Bigelow, and Kate Isaacs. Visualizing a moving target: A design study on task parallel programs in the presence of evolving data and concerns. *IEEE transactions on visualization and computer graphics*, 26(1):1118–1128, 2019.

[88] Moorthy Huck, Allan Porterfield, Nick Chaimov, Hartmut Kaiser, Allen Malony, Thomas Sterling, and Rob Fowler. An autonomic performance environment for exascale. *Supercomput. Front. Innov.: Int. J.*, 2(3):49–66, July 2015.

[89] Bibek Wagle, Mohammad Alaul Haque Monil, Kevin Huck, Allen D. Malony, Adrian Serio, and Hartmut Kaiser. Runtime adaptive task inlining on asynchronous multitasking runtime systems. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery.

[90] Parsa Amini and Hartmut Kaiser. Assessing the performance impact of using an active global address space in hpx: A case for agas. In *2019 IEEE/ACM Third Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM)*, pages 26–33. IEEE, 2019.

[91] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, pages 1–11, 2014.

[92] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 15–28, 2012.

[93] Nikunj Gupta, Jackson R Mayo, Adrian S Lemoine, and Hartmut Kaiser. Implementing software resiliency in hpx for extreme scale computing. *arXiv preprint arXiv:2004.07203*, 2020.

[94] Louisiana optical network initiative.

[95] F Incubator. Gloo: Collective communications library with various primitives for multi-machine training, 2017.