Journal of Big Data

RESEARCH

Open Access

Large scale performance analysis of distributed deep learning frameworks for convolutional neural networks

Marcel Aach[1,2*], Eray Inanc[1], Rakesh Sarma[1], Morris Riedel[1,2] and Andreas Lintermann[1]

*Correspondence:
m.aach@fz-juelich.de

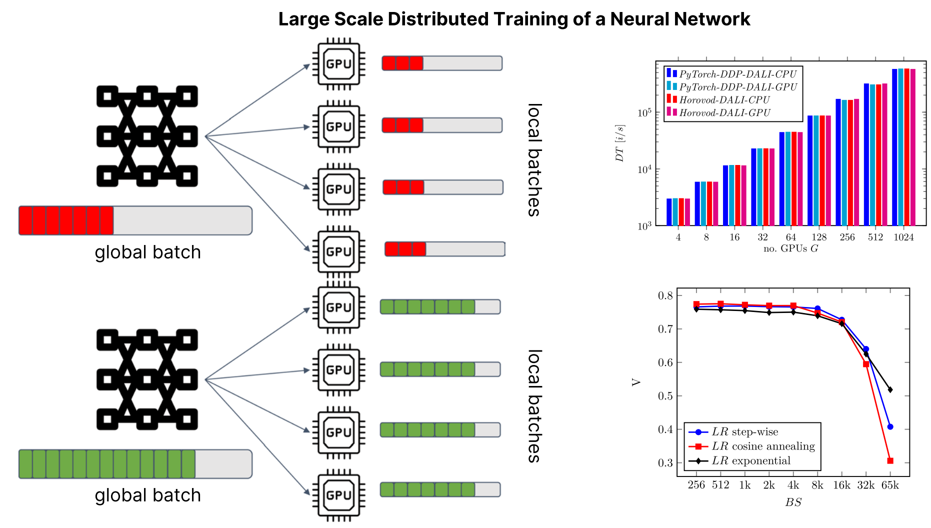
[1] Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Wilhelm-Johnen-Straße, 52428 Jülich, Germany
[2] School of Engineering and Natural Sciences, University of Iceland, Reikjavik, Iceland

Abstract

Continuously increasing data volumes from multiple sources, such as simulation and experimental measurements, demand efficient algorithms for an analysis within a realistic timeframe. Deep learning models have proven to be capable of understanding and analyzing large quantities of data with high accuracy. However, training them on massive datasets remains a challenge and requires distributed learning exploiting High-Performance Computing systems. This study presents a comprehensive analysis and comparison of three well-established distributed deep learning frameworks—*Horovod*, *DeepSpeed*, and *Distributed Data Parallel* by *PyTorch*—with a focus on their runtime performance and scalability. Additionally, the performance of two data loaders, the native *PyTorch* data loader and the *DALI* data loader by NVIDIA, is investigated. To evaluate these frameworks and data loaders, three standard ResNet architectures with 50, 101, and 152 layers are tested using the ImageNet dataset. The impact of different learning rate schedulers on validation accuracy is also assessed. The novel contribution lies in the detailed analysis and comparison of these frameworks and data loaders on the state-of-the-art Jülich Wizard for European Leadership Science (JUWELS) Booster system at the Jülich Supercomputing Centre, using up to 1024 A100 NVIDIA GPUs in parallel. Findings show that the *DALI* data loader significantly reduces the overall runtime of ResNet50 from more than 12 h on 4 GPUs to less than 200 s on 1024 GPUs. The outcomes of this work highlight the potential impact of distributed deep learning using efficient tools on accelerating scientific discoveries and data-driven applications.

Keywords: High-Performance Computing, Distributed deep learning, Performance analysis, Convolutional neural network, ImageNet

Graphical Abstract



Introduction

In the past few years, deep neural networks have become powerful tools to solve problems from different scientific disciplines. Especially in the field of image recognition, significant advancements have been made using Convolutional Neural Networks (CNNs) and Transformers [1, 2]. As the size of the datasets used for training and the model sizes continuously increase, their training becomes more and more computationally expensive. Therefore, it is of utmost importance to find suitable and efficient methods to reduce the training runtime by as much as possible.

Using High-Performance Computing (HPC) systems, it is possible to accelerate the training and model generation processes, i.e., by intelligently subdividing the problem and using massively parallel hardware for efficiently distributing the computational load. The two main strategies for distributing the training of neural networks to different workers, where a worker is usually a Graphics Processing Unit (GPU), are model and data parallelism [3]. The former method splits the neural network and distributes it across the workers. In contrast, the latter splits the input data and the network is trained with different batches per worker. At the end of an epoch, all gradients are merged to apply the same update to the network's weights on every worker. The Message Passing Interface (MPI) is frequently used to communicate the parameters between the workers in either a synchronous or asynchronous way. In the synchronous case, an `AllReduce` operation is executed for gradient reduction, while in the asynchronous case, a single central parameter server receives all gradient updates from the workers and performs the optimization step. For the asynchronous case, the overall performance is limited by the network bandwidth of the parameter server and depends on the amount of data to transmit per worker. The usage of alternative strategies such as asynchronous ring communications or employing MPI can alleviate this bottleneck. In general, with an increasing number of computational nodes, the communication to share the gradients becomes the main bottleneck. To minimize this communication overhead, the number of gradient reductions has to be reduced.

The focus of this study is on data parallelism, as it offers benefits for neural networks of any size when trained on large datasets, whereas model parallelism is primarily advantageous for network architectures that do not fit onto a single GPU. Consequently, data parallelism holds an advantage in its ability to cater to a wider range of neural network architectures, whereas model parallelism is better suited for handling larger neural networks that face memory limitations. The standard approach to scale Deep Learning (DL) trainings on large HPC systems is to increase the global batch size *BS*, which may, however, lead to insufficient validation accuracies [4]. To retain a sufficient accuracy, modifications to the training process are frequently necessary, which may also affect the parallel performance. It is furthermore challenging to apply the right framework for a specific learning task that involves large input data or models, and at the same time, benefit from the computational power of HPC systems.

The motivation of this study is to provide guidance in this direction. To this end, the performance in terms of accuracy and scalability of the parallel frameworks—*Horovod* [5], *PyTorch-Distributed Data Parallel* (*PyTorch-DDP*) [6] and *DeepSpeed* [7] are evaluated on the European HPC system Jülich Wizard for European Leadership Science (JUWELS) [8] Booster module (place 12 in Top500 [9] as of 08/2022). These three frameworks are easily accessible as they are open-source and have gained high popularity in recent years. Their analysis supports the research community to decide on a framework. The benchmarks are performed using the popular ImageNet dataset [10] and three residual neural networks (ResNets) with 50, 101, and 152 layers [1] to enable generic comparison with established benchmark studies.

The main contributions of this study are as follows:

- A comprehensive scalability analysis and comparison of training ResNet50, ResNet101, and ResNet152 on ImageNet with *Horovod*, *PyTorch-DDP*, and *DeepSpeed*, using the *DALI* and native *PyTorch* data loaders ranging on up to 1024 GPUs is performed. The results demonstrate that in combination with the native *PyTorch* data loader, *DeepSpeed* shows the best performance on a small number of GPUs, while *Horovod* and *PyTorch-DDP* outperform it when using a larger number of GPUs. However, in comparison, the use of *DALI* leads to higher throughput and improves scalability for all ResNet architectures.
- An assessment of the influence of step-wise, cosine, and exponential learning rate annealing on the validation accuracy for different batch sizes ranging from $BS = 256$ to 65,536 is performed. These findings reveal that cosine annealing delivers superior performance on small and medium batch sizes, while exponential annealing achieves the highest accuracy for the largest batch size.

The paper is structured as follows. The established distributed DL frameworks as well as the challenges that arise when scaling the training to large HPC systems are discussed in "Related work" section. In "Overview of the benchmark setup" section, an overview of the experimental setup is given. Subsequently, "Benchmark results" section presents the main benchmark results from a computational perspective. Finally, "Summary, conclusion, and outlook" section summarizes the findings, draws conclusions, and gives an outlook to future work.

Related work

The ImageNet dataset, originally introduced for the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), features 1,281,167 training images and 50,000 validation images divided into 1000 object classes [10]. This dataset is commonly used in the literature as an important benchmark to test algorithms, where their performance is measured through validation accuracy V [11], which is computed from correctly classified validation images.

AlexNet [12] (a CNN) was one of the first machine learning models to achieve high accuracy on the ImageNet dataset in the ILSVRC 2012. Since then, multiple improvements have been made to the original network structure [13], yielding the current ResNet architecture [1] with a varying amount of convolutional layers. Although Transformer architectures [14] have recently been shown to achieve even better results in image classification tasks, CNNs are still widely used due to lower training duration, low energy usage, and their good scalability [15, 16].

Many distributed DL frameworks exist for scaling especially neural network models to multiple workers in an HPC environment. A literature review is given in [17] and an in-depth performance analysis is presented in [3]. One of the first frameworks to scale to a large number of computational nodes equipped with Central Processing Units (CPUs) was *DistBelief* by Google [18] using an asynchronous Stochastic Gradient Descent (SGD) method. Others, such as *Petuum* [19] or *Project Adam* [20] have improved this idea, e.g., by introducing dynamic scheduling. With *FireCaffe* [21], the focus shifted towards using GPU clusters and synchronous gradient reduction methods.

The most common DL libraries such as *TensorFlow* [22] from Google, *PyTorch* from Facebook, and *MXNet* [23] from Apache, have their unique distributed training strategies. Other frameworks such as *HeAT* [24] take a more general approach to distributed machine learning by providing a scalable *NumPy*-like [25] Application Programming Interface (API) to enable all kinds of data analytics, not being limited only to neural networks. While all of these frameworks mainly focus on enabling data parallelism, more recently Microsoft introduced the *ZeRO* [26] and *DeepSpeed* [7] frameworks that can train models with billions of parameters through model parallelism. A small scale study (up to four GPUs) of the performance of *TensorFlow* and *FireCaffe* on different HPC systems is available in [27]. An overview of the frameworks and their parallelism and communication strategies are shown in Table 1.

The present work solely focuses on synchronous communication and data-parallel functionality of the commonly used frameworks. *Horovod*, *PyTorch-DDP*, and *DeepSpeed* are all compatible with the HPC system's job scheduler SLURM [28] and the InfiniBand communication pattern by default, while the distributed versions of *MXNet* and *TensorFlow* are not.

In the literature, various tests using a ResNet50 on the ImageNet dataset exist. In the original ResNet paper [1], it takes 29 h to train the network for 90 epochs on eight NVIDIA Tesla P100 GPUs with a batch size of $BS = 256$. Subsequently, the training time is reduced to 1 h on 256 NVIDIA Tesla P100 GPUs [4]. Finally, in [29], a training time of only 74 s on 2048 NVIDIA Tesla V100 GPUs is achieved. The current benchmark record is 28.8 s as of 10/2022 [30], where TPUs have been used instead of GPUs. To train a ResNet50 in such a short time, the authors increase the batch size to $BS = 8192$ and $BS = 81{,}920$. A large BS value, however, usually leads to a reduction in generalization performance. To prevent this, several hyperparameter modifications are applied in [29] to reach a V comparable to that of [1]. Therefore, the learning rate LR is scaled linearly with respect to the number of workers. This is motivated by the fact that with fewer weight updates, the learning rate has to be increased to achieve similar gradient adjustments compared to using a small learning rate with more frequent gradient updates. However, this linear scaling rule does not apply to all cases, i.e., during the start of the training, where a lot of parameters are subject to changes, a large learning rate may prohibit the optimizer from converging. Therefore, a warm-up technique is used that slowly increases the learning rate during the first five epochs [31]. Another technique reported to improve the accuracy when training with a large BS is label-smoothing [32], which is a regularization method adapted to classification models. Using larger ResNet architectures, such as the ResNet101 or ResNet152, leads to slightly improved V values [1].

Code Snippet 1 Integration of *Horovod*

```
import torch
import horovod.torch as hvd

# Initialize Horovod
hvd.init()

# Define model and optimizer
model = models.ResNet50()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Add Horovod distributed optimizer
optimizer = hvd.DistributedOptimizer(optimizer,
            named_parameters=model.named_parameters())

# Broadcast model to processes
hvd.broadcast_parameters(model.state_dict(), root_rank=0)

# Start training loop
for epoch in range(100):
...

# Clean-up the parallel process group
hvd.shutdown()
```

In this work, the learning rate warm-up, learning rate scaling, and the label-smoothing techniques are used to stabilize the training with the default SGD optimizer with

Table 1 Overview of distributed DL frameworks, adapted from [6, 17]

Framework	Parallelism	Communication
DistBelief [18]	Model + Data	Asynchronous
FireCaffe [21]	Data	Synchronous
Horovod [5]	Model + Data	Synchronous
MXNet [23]	Model + Data	Bounded Asynchronous
Petuum [19]	Model + Data	Bounded Asynchronous
TensorFlow [22]	model + Data	Bounded Asynchronous
PyTorch-DDP [6]	Model + Data	Synchronous
DeepSpeed [7]	Model + Data	Synchronous

Bounded asynchronous is a hybrid of synchronous and asynchronous communication

relatively large *BS* values. Additionally, three different learning rate schedules are explored and their performance in terms of *V* is analyzed.

Overview of the benchmark setup

This section gives an overview of the main frameworks used in this study, i.e., *Horovod* is introduced in "Horovod" section, *PyTorch* in "PyTorch-distributed data parallel" section, *DeepSpeed* in "DeepSpeed" section, and the data loaders in "Data loaders and data-set compression" section. Furthermore, their communication operations are presented and examples of how to include them into actual Python code are provided. General issues that arise when scaling to a large amount of GPUs are addressed in "GPU scaling issues" section and the different ResNet architectures are introduced in "Residual neural networks" section. Three different learning rate scheduling methods with the potential of increasing the accuracy of the training are introduced in "Learning rate scheduling" section. The hardware and software configuration of the supercomputer used for the benchmark tests is presented in "JUWELS HPC system and software stack" section.

Horovod

Horovod is an open-source distributed DL library originally developed by Uber for *TensorFlow* [5]. It is also supported as a backend library in the most common DL frameworks such as *PyTorch* and Apache *MXNet*. Minimal code changes are required to integrate *Horovod* into these DL frameworks. Code snippet 1 gives an example of how to integrate *Horovod* with *PyTorch*.

The work by Pumma et al. [33] provides an overview and an analysis of the communication patterns in *Horovod*. It is one of the first libraries to use a decentralized `Ring AllReduce` approach [34] to compute the gradient reduction instead of a single parameter server receiving all the updates, cf. "Introduction" section. It relies on low-level communication libraries such as *MPI*, the *NVIDIA Collective Communications Library (NCCL)* [35], or Facebook *Gloo* [36]. It is observed that the *NCCL* `AllReduce` yields superior performance on NVIDIA GPUs [6].

On a local worker level, the communication operations in *Horovod* are asynchronously handled by a separate background thread. This thread repeatedly checks for communication requests and performs the corresponding data transfers. Since these transfers may be requested asynchronously, the order of execution per worker may also be different. However, *Horovod* uses collective communication directives from other libraries and hence has to execute a consensus protocol to ensure consistency (in terms of order) across all workers. The process is summarized here:

(i) One global background thread receives all the transfer requests from the local background threads.
(ii) The global background thread puts the requests in the correct order and sends the list back to the local instances.
(iii) Each local thread combines its local data and carries out the data exchange with the other workers via `AllReduce`.

This back and forth communication creates overhead that can limit the scalabilty of the framework.

In *Horovod*, the computations and communications are coupled with the ability to batch small `AllReduce` operations. This exploitation of batching communication operations is known as tensor fusion [5]. With this operation, the smaller data volumes are transferred across different workers by locally fusing the data that are ready to be reduced. Hence, fewer `AllReduce` operations are required. In large neural networks with large number of parameters, this operation is expected to yield huge parallel performance gains.

PyTorch-distributed data parallel

PyTorch is a machine learning framework primarily developed by Facebook AI Research. The *PyTorch-DDP* module features a built-in way to parallelize the training of neural networks across multiple workers, e.g, GPUs. Code snippet 2 shows an example of how *DDP* in *PyTorch* is used. Similar to *Horovod*, the *PyTorch-DDP* library uses an `AllReduce` paradigm (with the communication libraries *NCCL*, *Gloo*, or *MPI*) for updating the gradients used in deep neural networks. To trigger the communication operation, a custom ‘hook’ is registered in the internal automatic differentiation engine that is integrated into the backward pass operation of deep neural networks [6]. A separate code for managing the communication is hence not required.

Analogous to *Horovod*’s tensor fusion operation, *PyTorch-DDP* features gradient bucketing, where instead of an immediate `AllReduce` operation the algorithm waits for a few processor cycles once a batch of gradients is complete, and buckets (or ‘fuses’ in the sense of *Horovod*) multiple gradient parameters into a single parallel operation. Hence, the computation and communication are overlapped, thus skipping frequent gradient synchronization. A drawback of this method is a possible mismatch in the `AllReduce` operation if the reduction order is not the same across all processes—resulting in an incorrect reduction or data inconsistencies. This issue is addressed by bucketing the gradients in the reversed order obtained during the forward pass operation. This is motivated by the fact that the last layers of a network are likely the first ones to finish computation during the backward pass. Another issue is the skipped bucketed gradients that never enter the `AllReduce` operation. *PyTorch-DDP* handles this issue by a participation algorithm, which checks the output tensors during the forward pass to find all non-participated parameters (i.e., based on gradients that have not been updated) in the current iteration to be included in the next iteration.

Code Snippet 2 Integration of *PyTorch-DDP*

```
import torch.distributed as dist
import torch.nn.parallel as par

# Initialize the parallel process group
dist.init_process_group(...)

# Define model and optimizer
model = models.ResNet50()
opt = optim.SGD(model.parameters(), lr=0.01)

# Distribute model to processes
dist_model = par.DistributedDataParallel(model)

# Start training loop
for epoch in range(100):
...

# Clean-up the parallel process group
dist.destroy_process_group()
```

DeepSpeed

The focus of *DeepSpeed* developed by Microsoft Research is on training large language models. These models usually feature several billion parameters and are trained on datasets from the natural language domain, which are significantly larger than most computer vision datasets. The main issue with these large language models is their massive memory footprint, a problem that is addressed with the *ZeRO* optimizer as part of *DeepSpeed*. This parallel optimizer eliminates memory redundancies by not only distributing the training data across workers but also the optimizer, gradient, and (if required) model parameters across workers. In contrast to the default data-parallel approach, the model is, therefore not necessarily replicated on each worker. Still, after each training step, an `AllReduce` communication step is necessary to ensure consistency. This is performed in a two-step process: first, different parts of the data are distributed to different workers with a `ReduceScatter` command, then each worker gathers the different chunks of data with an `AllGather` operation [26]. Code snippet 3 shows the integration of *DeepSpeed* within *PyTorch*, which is currently the only supported DL backend.

Code Snippet 3 Integration of *DeepSpeed*

```
import torch
import deepspeed as ds

# Initialize the parallel process group
ds.init_distributed(...)

# Define model and optimizer
model = models.ResNet50()
opt = optim.SGD(model.parameters(), lr=0.01)

# Distribute model and optimizer
distrib_model, distrib_optimizer = ds.initialize(
                                    model=model,
                                    optimizer=optimizer,
                                    model_parameters=model.parameters())
# Start training loop
for epoch in range(100):
...

# Clean-up the parallel process group
ds.sys.exit()
```

Data loaders and dataset compression

Two types of data loaders and corresponding dataset compressions are compared in this work. One of the data loaders is the native *PyTorch* data loader, which uses the raw ImageNet data in the `JPG` format. This data loader only supports raw image data and performs all pre-processing steps on the CPUs. The other data loader is the *NVIDIA Data Loading Library (DALI)* [37], where a compressed version (`TFRecord`) of the ImageNet dataset is used. *DALI* is an open-source framework to accelerate the data-loading process in DL applications by involving the GPU, following a pipeline-based approach. Usually, the GPU runs computations much faster compared to the data-loading speed of the CPU. The idea of DALI is to prevent the GPU from starving by moving the data-loading process to the GPU at an early stage. The GPU then performs the data pre-processing, such as image resizing, cropping, and normalization on the fly. By pipe-lining these operations and executing them directly on the GPU, *DALI* minimizes the amount of data that needs to be transferred between the CPU and GPU, which reduces the overhead associated with these operations. *DALI* supports multiple data formats and with its unified interface, it is easy to integrate into all common DL frameworks. With this seamless integration developers can exploit the full potential of their GPU-based systems without having to modify their existing workflows significantly or switch between different data loading libraries. While the main focus of *DALI* is the GPU-based approach, it also offers the possibility to use the CPU for all steps of the pipeline. In this case also the pre-processing is performed on the CPU. Initial benchmarks show a speed-up between 20-40% in throughput compared to the original *PyTorch* data loader [38].

It should be noted that in terms of actual disk space, the difference between the compressed `TFRecord` version of the ImageNet dataset (144 GB) and the raw `JPG` data (154 GB) is marginal. However, the file structure of the `TFRecord` dataset is much better suited for data loading in comparison to the over one million single image files in the raw dataset.

GPU scaling issues

GPU scaling in deep learning presents several challenges, including communication overhead, load balancing, and memory limitations. Communication overhead arises due to the constant synchronization and information exchange required between multiple GPUs during training. This overhead can reduce efficiency and performance as it grows with the number of GPUs. Solutions include using high-bandwidth, low-latency interconnects, and implementing efficient communication algorithms such as ring-based `AllReduce` methods [39, 40]. Load balancing is crucial for ensuring an even distribution of computational workload across all GPUs, maximizing resource utilization. An uneven workload distribution can lead to idle GPUs, wasting resources and increasing runtime. Dynamic load balancing algorithms and data or model parallelism techniques can help distribute tasks and data efficiently across multiple GPUs. Memory limitations pose a challenge when large models (or datasets) exceed a single GPU memory capacity, causing out-of-memory errors or forcing smaller batch sizes, which can negatively impact performance and convergence.

Residual neural networks

A prevalent challenge when training deep neural networks is the vanishing gradient problem, which leads to accuracy degradation [41]. ResNet architectures address this issue by introducing "skip-connections" that enable the training of deep networks without compromising accuracy. For larger vision datasets, ResNet50, ResNet101, and ResNet152 are the most widely adopted models. All models consist of one input layer and one fully-connected output layer but vary in the number of intermediate convolutional layers (48 vs. 98 vs. 150), see Fig. 1 for a visualization of the architecture. As a result, 3.8×10^9 floating-point operations per forward pass are needed for a ResNet50, 7.6×10^9 for a ResNet101, and 11.3×10^9 for a ResNet152. Although having more layers typically allows for the representation of more complex phenomena, it is essential to

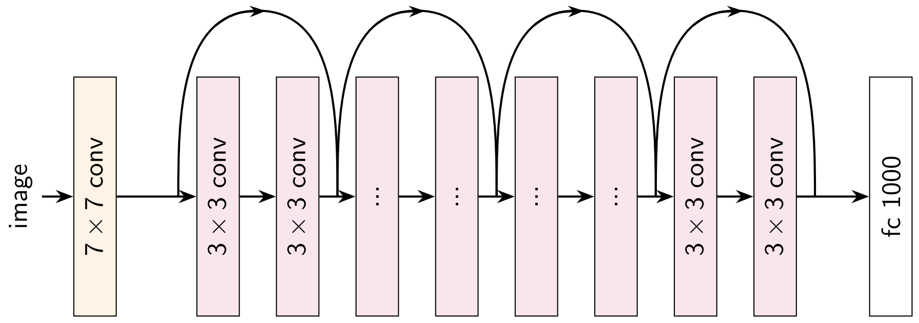


Fig. 1 Standard ResNet architecture with one input layer (in orange), a varying number convolutional layers (in purple) and a fully-connected output layer (in white). Figure adapted from [1]

consider the trade-off between model complexity and training efficiency, as an increased number of floating-point operations leads to longer runtimes.

Learning rate scheduling

A known problem in large-scale distributed DL is the major drop in the validation accuracy V when using a large BS [4], regardless of the used framework, data loader, or optimizations (e.g., label-smoothing). A larger BS yields fewer optimization steps, thus compromising the accuracy of the optimizer. This issue is one of the key challenges in distributed DL. To avoid divergence of the training process for $BS \geq 32k$ on ImageNet, further optimizations, such as LR scheduling, are necessary. The most common approach for a LR schedule on the ImageNet dataset is a stepwise annealing method [1]. This schedule reduces the LR in regular intervals over time by an order of magnitude, i.e., in the ImageNet training, these intervals are set at epoch numbers 30, 60, and 80.

Since the LR schedule plays an important role in the performance of a model, different approaches exist in the literature. The cosine-annealing schedule [42] is supported by *PyTorch* and uses the cosine function for smoother LR annealing over time. In this case, the learning rate LR_t at the current epoch t is defined by:

$$LR_t = LR_{min} + \frac{1}{2}(LR_{max} - LR_{min})\left[1 + \cos\left(\frac{t}{t_{max}}\pi\right)\right], \tag{1}$$

where LR_{min} and LR_{max} are the minimum and maximum values of the learning rate, and t_{max} is the total number of epochs used in the training. This decays LR gradually at every epoch, compared to the sharp drops of the step-wise annealing. Another approach to LR scheduling is the exponential decay schedule. Here, the LR starts with a large value and is then decreased rapidly in the beginning and gradually afterwards in an exponential manner. The learning rate LR_t at the current epoch t is given by:

$$LR_t = LR_{max} * \gamma^t, \tag{2}$$

where the decay factor is usually set to $\gamma = 0.95$. The comparison in Fig. 8a shows that this scheduling method decreases the learning rate at similar orders of magnitude as the step-wise scheduler but in a smoother way.

JUWELS HPC system and software stack

The benchmarks presented in "Benchmark results" section are performed on the JUWELS HPC system. This system has a Modular Supercomputing Architecture (MSA) [43] and consists of a cluster and a booster module. The experiments use the GPU-based JUWELS Booster module, which consists of 936 compute nodes, each equipped with two AMD EPYC Rome 7402 CPU with 2x24 cores, clocked at 2.8 GHz, 512 GB Dynamic Random Access Memory (DRAM), and four NVIDIA A100 GPU with 40 GB High Bandwidth Memory (HBM). The GPUs communications in a compute node are performed via NVLINK [44]. The interconnect between compute nodes is a Mellanox InfiniBand HDR network with DragonFly+ topology. Each compute node has four HDR host channel

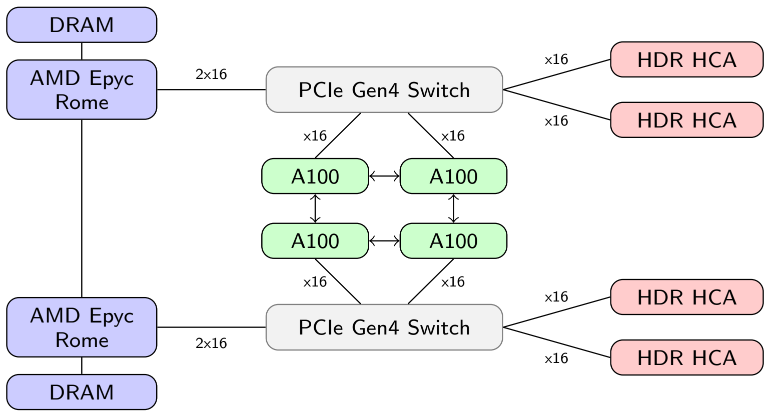


Fig. 2 JUWELS Booster node schematic. Two AMD Epyc Rome CPUs are connected to four NVIDIA A100 GPU and the HDR HCAs via two PCIe Gen4 switches. The GPUs communicate via NVLINK

adapters. A Peripheral Component Inter-Connect Express (PCIe) Gen4 bus connects the components. In total, the JUWELS Booster is equipped with 3,744 GPUs and has 73 PFlops peak performance. Figure 2 shows the schematic of a single node.

The compressed and uncompressed ImageNet datasets are both stored on the *SCRATCH* partition of the JUWELS General Parallel File System [45]. This partition is optimized for the storage of large data and features a high Input/Output (I/O) bandwidth.

For running the experiments, the following software versions are deployed, which are available through JUWELS' EasyBuild [46] software system:

- `GCC 11.2.0`
- `OpenMPI 4.1.2`
- `Python 3.9.6`
- `CUDA 11.5`
- `PyTorch 1.11.0`
- `Horovod 0.24.3`
- `Deepspeed 0.6.3`
- `DALI 1.12.0` (virtual environment)

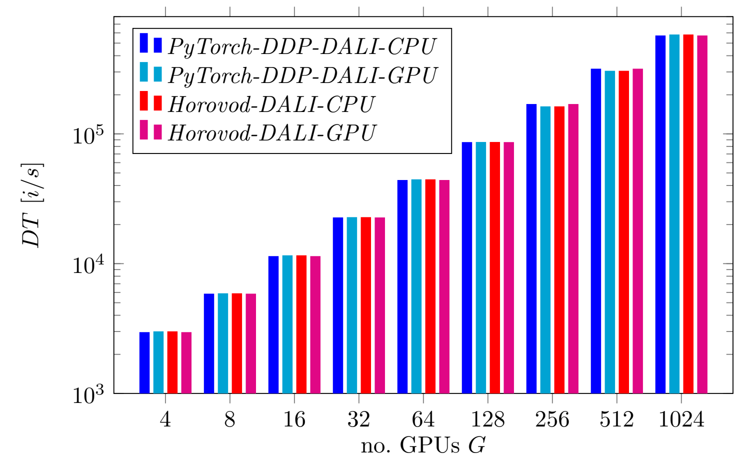
The number of CPU threads per data loader instance is set to 8.

Benchmark results

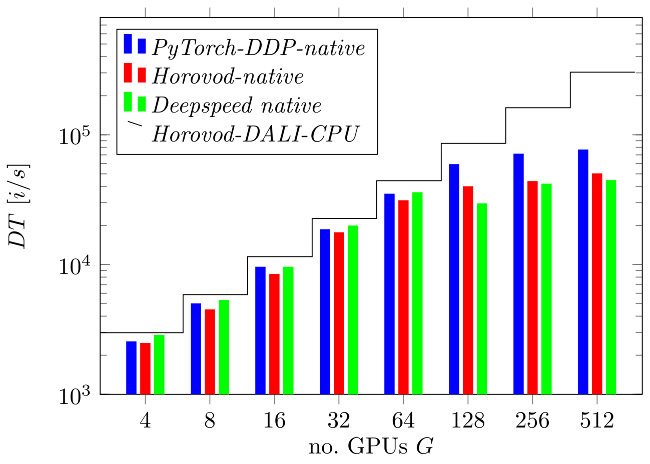
This section evaluates the performance of the three frameworks *Horovod*, *PyTorch-DDP*, and *DeepSpeed* with the native *PyTorch* and the *DALI* data loaders on the JUWELS Booster. The runtime T of training a ResNet50, a ResNet101, and a ResNet152 on the ImageNet dataset for 90 epochs with a batch size of $BS = 64$ per GPU is measured and analyzed in "Efficiency" section. Additionally, the effect of three different learning rate schedulers on V is explored in "Accuracy" section.

Efficiency

The results for the ResNet50 training in terms of data throughput DT, measured in images i per second over the number of GPUs, are shown in Fig. 3. Overall, the *DALI*



(a) Throughput of *Horovod* and *PyTorch* with the ***DALI* data loader** CPU and GPU version on the **compressed** ImageNet dataset.



(b) Throughput of *Horovod*, *PyTorch-DDP*, and *DeepSpeed* with the **native** *PyTorch* data loader on **raw** ImageNet dataset, including comparison with *Horovod-DALI-CPU* throughput. The largest configuration only features 512 GPUs in this case as no significant additional speed-up is expected on larger configurations.

Fig. 3 Throughput of different frameworks and *DALI* (**a**) and native (**b**) data loader for the **ResNet50** case, averaged over three experimental runs. The variance between runs is small (in general < 5%) and therefore not shown

data loader (Fig. 3a) achieves a higher throughput of images compared to the native *PyTorch* data loader (Fig. 3b), and this is observed to be independent of the distributed DL framework. For the native data loader, the three frameworks show similar performances up to 64 GPUs. For a smaller number of GPUs, *DeepSpeed* shows the highest *DT*, but performance drops for a larger number of GPUs, where *PyTorch-DDP* performs the best. In summary, for all three frameworks, it is evident that the native data loader cannot match the performance of the *DALI* data loader.

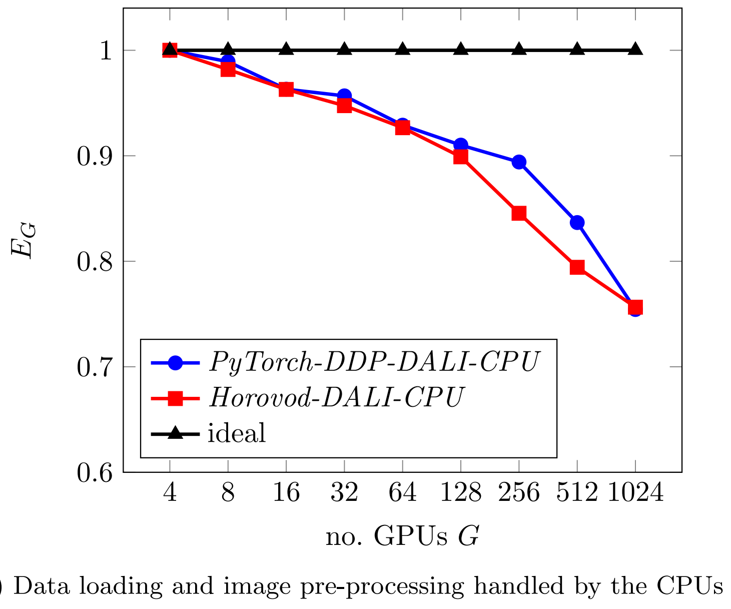
For a consistent scalability comparison, the parallel efficiency metric E_G is calculated as

$$E_G = \frac{S_G}{G}, \tag{3}$$

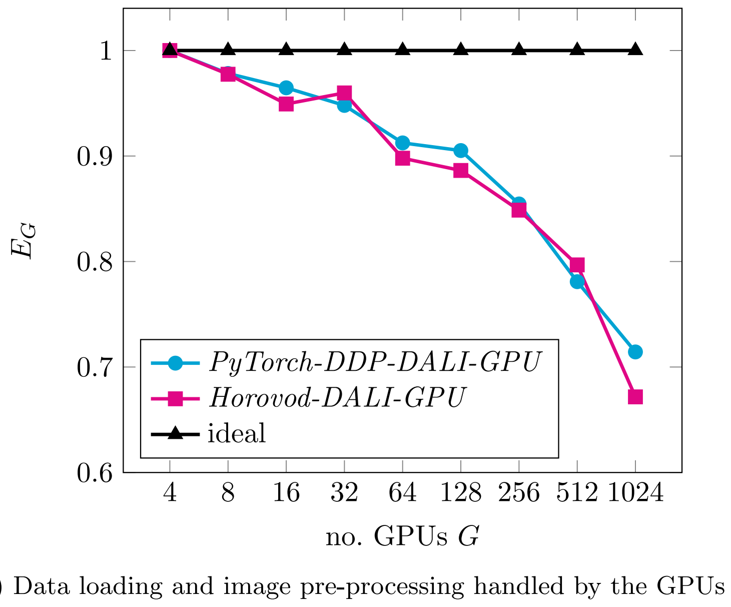
where S is the speed-up and G is the number of workers. The speed-up S_G is computed as

$$S_G = \frac{T_4}{T_G}, \tag{4}$$

with reference runtime T_4, i.e., using four GPUs (one node on the JUWELS Booster). Note that E_G of (or close to) unity is the ideal scenario with perfect scaling. The quantity E_G is plotted in Fig. 4 for the *DALI* data loader over the number of GPUs. Note that



(a) Data loading and image pre-processing handled by the CPUs



(b) Data loading and image pre-processing handled by the GPUs

Fig. 4 Parallel efficiency of *Horovod* and *PyTorch-DDP* on up to 1024 GPUs with the *DALI* data loader for CPU- (**a**) and GPU-based (**b**) pre-processing with compressed ImageNet dataset for the **ResNet50** case, averaged over three runs. Black line denotes the ideal case. The variance between runs is small (in general $< 5\%$) and therefore not shown

the *DeepSpeed* framework cannot use the *DALI* data loader. It is clear that the tested frameworks show similar scaling performances. Independent of the hardware acceleration (pre-processing on CPU in Fig. 4a or GPU in Fig. 4b) and the framework, the E_G value remains above 0.65 up to 1024 GPUs. The training with *PyTorch-DDP* using the CPU for data input performs slightly better than its *Horovod* counterpart on 256 and 512 GPUs. With 1024 GPUs, the trainings using CPU for data input achieve a higher E_G value of 0.76 compared to the ones using GPU, which is at $E_G = 0.68$. These findings show data loading with CPUs to be favorable for large-scale trainings. An analysis of the average CPU usages shows an occupancy of less than 40% across all configurations. It is assumed that the computationally strong host CPUs make up for any performance gains achieved by transferring the image pre-processing onto the GPUs. For hardware setups with less powerful host CPUs, using the GPU-based *DALI* version could still improve performance.

Figure 5 presents the parallel efficiency results for the native *PyTorch* data loader. When compared to the scalability of the *DALI* data loader (see Fig. 4), the scaling performance of the tested frameworks is considerably worse. The E_G values of the training with *PyTorch-DDP* using the native data loader (denoted as *PyTorch-DDP-native* in Fig. 5) drops below 0.44 already with 256 GPUs, whereas *Horovod* and *DeepSpeed* have E_G values of 0.50 and 0.33, respectively on 128 GPUs. With 512 GPUs, all of the frameworks achieve an E_G value of less than 0.24, indicating the limitations of the data loader on parallelization. The superior performance of the *PyTorch-DDP-native* data loader could be due to its better compatibility with the *PyTorch-DDP* framework.

Fig. 6 shows the scaling performance of three ResNet architectures using the *DALI* data loader and *PyTorch-DDP* framework. The ResNet50, 101, and 152 model show similar E_G values up to 16 GPUs. On larger configurations with more GPUs, the ResNet152 achieves the highest E_G values, reaching 0.81 on 1024 GPUs. When more than 512 GPUs are utilized, the ResNet50 achieves the lowest E_G values. This behavior is expected, as

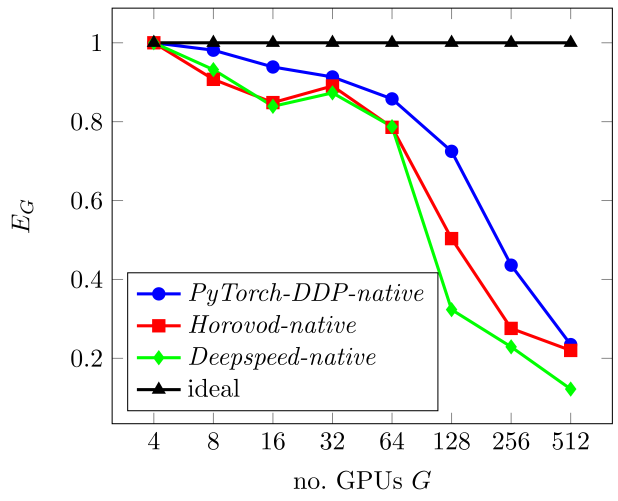


Fig. 5 Parallel efficiency of *Horovod*, *PyTorch-DDP* and *DeepSpeed* on up to 512 GPUs with the native *PyTorch* data loader and raw ImageNet dataset for the **ResNet50** case, averaged over three runs. Black line denotes the ideal case. The variance between runs is small (in general $< 5\%$) and therefore not shown

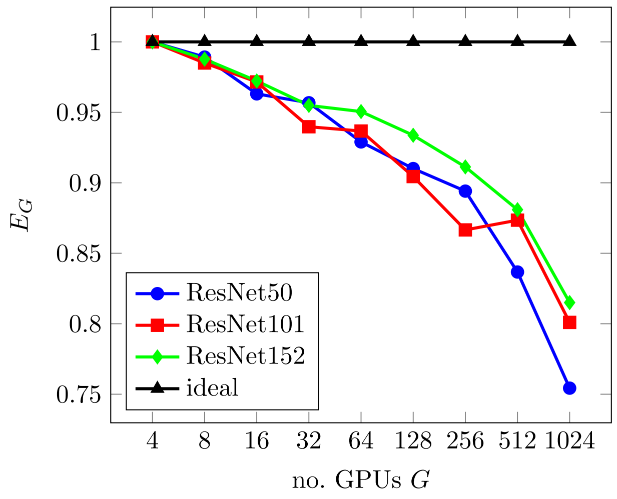


Fig. 6 Parallel efficiency comparison of *PyTorch-DDP* on up to 1024 GPUs for **different ResNets** with *DALI* data loader (CPU-based) and compressed ImageNet dataset, averaged over three runs. The black line denotes the ideal case. The variance between runs is small (in general $<$ 5%) and therefore not shown

a larger neural network means that more computation is necessary, which improves the computation to communication ratio and therefore also the scaling behavior. Nevertheless, the superiority of the *DALI* data loader over the *native* data loader can also be observed for the training with ResNet101 (see Fig. 7). On configurations with more than 32 GPUs, the *DALI* data loader clearly outperforms the *native* one in terms of scaling performance. Moreover, it is evident that *PyTorch-DDP* shows slightly better scaling performance than the *Horovod* framework (compare blue with green lines in Fig. 7).

For further verification of the results and a comparison of the *DALI* and native data loader, the NSys profiling tool [47] is used to analyze the amount of communication,

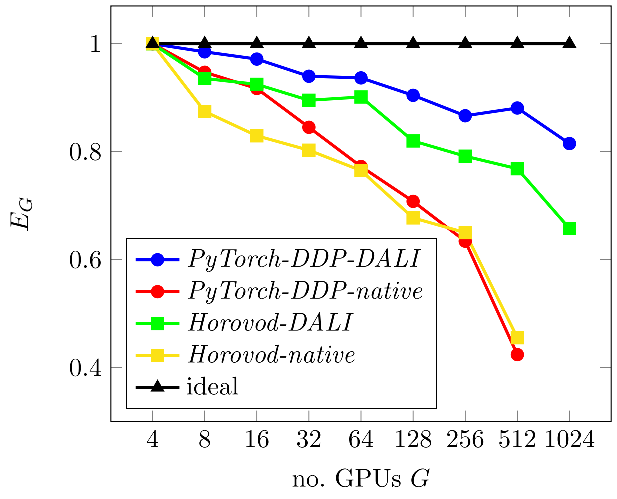


Fig. 7 Parallel efficiency of *Horovod* and *PyTorch-DDP* on up to 1024 GPUs training a **ResNet101** with the *DALI* data loader and compressed ImageNet dataset and native *PyTorch* data loader and uncompressed ImageNet dataset, averaged over three runs. Black line denotes the ideal case. The variance between runs is small (in general $<$ 5%) and therefore not shown

computation, and data loading that takes place during the training runs of all three ResNet architectures, where the results are shown in Table 2 in terms of three quantities. These are: (i) the runtime share by the Compute Unified Device Architecture (CUDA) kernels to perform 'communication'-related NCCL tasks (named communication), (ii) the runtime share to perform 'computation'-related tasks with the *cuDNN* library [48] (named computation), such as the time for the calculation of the convolutional layers, and (iii) the runtime share to execute 'input/output'-related tasks, such as data loading (named I/O).

For both types of data loaders, the time percentage spent on communication increases with the number of workers, while the efforts for the computation and data loading processes reduce. This behavior is expected when scaling up tasks that require frequent communication on a cluster while keeping the size of the dataset

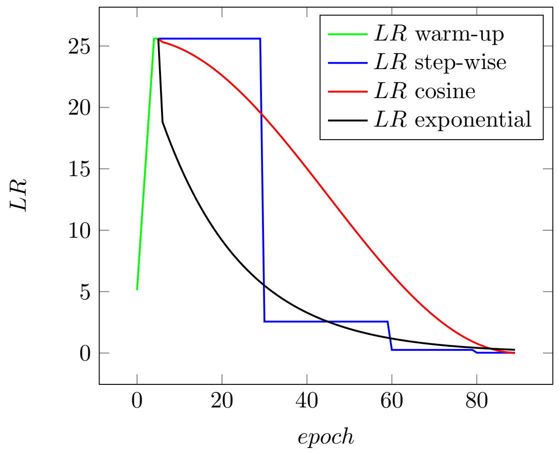
Table 2 Profiling CUDA kernel time in percent spent on communication operations via `AllReduce`, computations with the *cuDNN* library, and data loading functions

No. GPUs	***PyTorch-DDP DALI***			***PyTorch-DDP native***		
	AllReduce [%] (Communication)	**data[%] (I/O)**	***cuDNN*[%] (Computation)**	**AllReduce [%] (Communication)**	**data[%] (I/O)**	**cuDNN[%] (Computation)**
(a) Training of ResNet50 on ImageNet						
4	15.40	22.00	32.50	22.40	21.00	30.80
8	19.00	21.40	31.75	23.95	20.05	29.20
16	21.00	20.95	30.70	27.15	18.83	27.35
32	27.09	18.98	28.14	31.30	17.26	25.11
64	30.87	17.76	26.35	32.75	16.30	23.55
128	33.61	17.03	24.99	49.48	11.77	17.33
256	37.08	15.78	23.26	76.77	5.06	7.14
512	43.48	13.57	20.02	82.61	3.66	5.52
1024	46.18	11.56	17.31	–	–	–
(b) Training of ResNet101 on ImageNet						
4	13.30	23.00	46.00	28.65	22.50	38.12
8	20.55	21.25	41.45	30.15	18.28	35.52
16	24.08	20.37	39.65	35.67	16.76	32.71
32	25.36	18.71	36.99	35.46	14.59	28.43
64	37.17	16.69	33.39	37.69	15.31	29.88
128	36.29	16.74	34.02	42.32	13.39	26.38
256	39.31	15.54	31.56	56.43	11.38	22.83
512	37.73	15.40	31.59	59.18	11.87	24.45
1204	49.18	11.87	24.45	–	–	–
(c) Training of ResNet152 on ImageNet						
4	16.20	22.40	44.60	18.41	21.97	44.17
8	20.55	21.75	42.35	20.65	21.95	40.75
16	25.90	20.05	39.07	24.77	20.70	38.62
32	29.16	18.72	37.15	30.31	18.77	35.32
64	33.56	16.90	33.82	38.34	16.42	30.80
128	36.16	16.66	33.73	45.75	14.02	26.60
256	38.33	15.51	31.60	49.39	15.05	28.46
512	40.36	14.41	29.43	51.76	11.16	25.36
1024	43.21	13.08	26.99	–	–	–

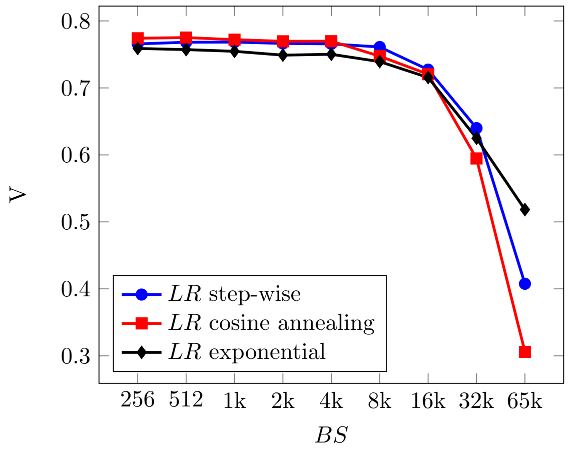
The first 10 epochs of the training process are profiled with the NSys Profiler (first five epochs for four GPUs due to time limits of the profiler) constant. For a smaller number of GPUs, the difference in time spent executing the `AllReduce` commands is similar for the *DALI* and native data loader methods. However, this difference increases rapidly with more workers. For example, in the ResNet50 case with 512 GPUs, the native data loader spends 82.61% of its time on communication, while the *DALI* data loader spends only 43.48%. A similar trend can be observed for the trainings with ResNet101 and ResNet152, where the native data loader spends 59.18 and 51.76% of its time on communication, compared to 37.73% and 40.36 for the *DALI* data loader, respectively. This substantial discrepancy could explain the poor scaling behavior of the native data loader. Regarding the computation time with the *cuDNN* library, it decreases for both data loaders as the number of GPUs increases, which is expected as the overall computational workload is distributed across a larger number of GPUs. For all three ResNet cases, the *DALI* data loader consistently exhibits higher computation percentages than the native data loader, suggesting that it effectively utilizes GPU resources. As for data loading, the time spent decreases as the number of GPUs increases for both data loaders. Although the relative data loading time is comparable between the two data loaders, it is important to emphasize that the *DALI* data loader is much faster in absolute timing. For example, in the ResNet152 case on 64 GPUs, the *DALI* data loader is responsible for 16.9% of the total runtime which amounts to $\approx 25s$ in absolute timing. For the native data loader case, the relative value is roughly the same with 16.42% of the total runtime, which, however, amounts to $\approx 47s$ in absolute timing. As expected, when comparing the three ResNet models it is evident that the communication overhead slightly reduces for smaller ResNet architectures, while the computation time increases as the size of the ResNet grows. Due to the low scaling performance of the native data loader, no evaluations on 1024 GPUs were performed for this case.

Accuracy

To investigate the issue of lower accuracy with a larger BS value, the effect of different learning rate schedules on the learning rate LR itself and V are explored in Fig. 8 for the training with ResNet50. The three methods deployed in this case are thestep-wise, the cosine, and the exponential annealing methods, as described in "Learning rate scheduling" section. Fig. 8b depicts the evaluation of V using the different learning rate schedules over growing batch sizes. All three scheduling methods have similar performances up to $BS = 4k$ (corresponds to 64 GPUs) with the exponential scheduling method being slightly worse than the others. At $BS = 8k$ (128 GPUs), the first significant drop of V from $\approx 77\%$ to 74% is observed. From $BS = 16k$ (256 GPUs), the quantity V drops consistently to lower values. For all three scheduling methods, there is a sharp drop of V for $BS = 32k$ and $BS = 65k$ (512 and 1024 GPUs). The difference in V is large for $BS = 65k$, where $V \approx 52\%$, $V \approx 41\%$, and $V \approx 31\%$ for the exponential annealing, step-wise, and cosine annealing scheduling methods, respectively. It is interesting to observe that the exponential scheduling method outperforms the cosine annealing for $BS = 32k$ and also the step-wise scheduling method for $BS = 65k$, even though exponential scheduling starts with the worst V value even for $BS = 256$. It is



(a) Learning rate LR variation over epochs using different learning rate scheduling methods including warm-up in first five epochs (on 1,024 GPUs).



(b) Validation accuracy V for different learning rate schedulers with increasing batch size BS. Average over 3 runs.

Fig. 8 Analysis of different learning rate schedulers for **ResNet50** training, showing learning rate over epochs (**a**) and validation accuracy over batch size (**b**). Note the original learning rate *LR* of 0.025 is scaled with the number of GPUs

evident that none of the scheduling methods can avoid the drop in V for trainings with large BS values, however, a training with the exponential learning rate schedule is the most favorable for large BS. Figure 9 depicts the validation accuracy curves over the number of epochs for an exemplary training of a ResNet50 with $BS = 65k$ with the three different learning rate schedules. While the learning curves show similar behaviour for the first 20 epochs, the exponential schedule outperforms the other two in the following 70 epochs by large margin.

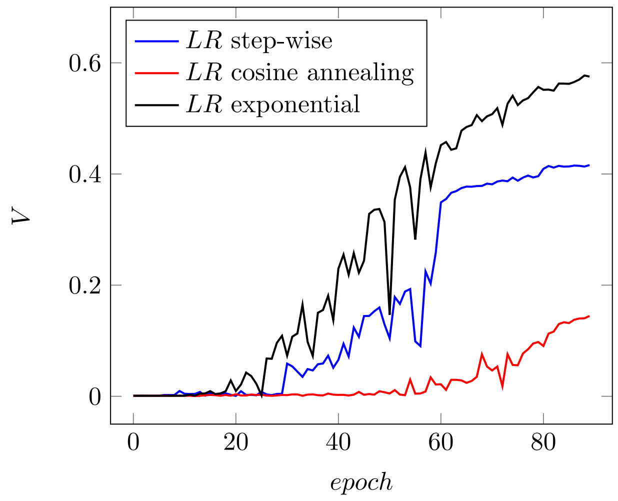


Fig. 9 Validation accuracy V over the number of epochs for a ResNet50 training with $BS = 65k$ on 1024 GPUs with different learning rate schedulers

Summary, conclusion, and outlook

In this study, three distributed DL frameworks, i.e., *PyTorch-DDP*, *Horovod*, and *DeepSpeed* were analyzed in combination with the *DALI* and native *PyTorch* data loader on the JUWELS Booster module on up to 1024 GPUs in terms of data throughput, the runtime, and the scaling performance. For this analysis, the ResNet50, ResNet101, and ResNet152 architectures were trained on the ImageNet dataset. Furthermore, the impact of the batch size on the validation accuracy and the effect of different learning rate scheduling methods were investigated for training a ResNet50.

The superiority of the *DALI* data loader over the native framework-based data loader in terms of scaling performance was evident. A parallel efficiency of over 0.85 on up to 256 GPUs and over 0.75 on 1024 GPUs for training ResNet50 was achieved. This value was over 0.80 on 1024 GPUs for training ResNet101 and ResNet152. It can be concluded that *DALI* is well suited to be used in large-scale distributed machine learning setups, regardless of the underlying framework or size of the neural network. Comparatively, the native *PyTorch* data loader could only achieve an efficiency of 0.45 for a training with 512 GPUs, hence, an even lower number of GPUs.

As the global batch size has to be increased with the number of GPUs, the good scaling performance can only be reached with a large global batch size, leading to a reduction in validation accuracy of the training. Even though no solution exists to address this problem, this work has shown that some mitigation was possible through choosing the right learning scheduling methods. An exponential learning rate scheduling method showed the best performance in terms of validation accuracy for a large batch size of 65k on 1024 GPUs for training ResNet50, whereas for smaller batch sizes, the cosine or step-wise annealing scheduling methods achieved better accuracies.

Overall, the total training time was reduced from $\approx$ 13 h on 4 GPUs to $\approx$ 200 s on 1024 GPUs for training ResNet50 (234 times faster) and $\approx$ 17 h to $\approx$ 300 s for the Resnet152 case (204 times faster), respectively. This good scaling behavior proves the combined power of distributed DL and HPC when using the right tools and methods. Such short training times enable the developers to focus more on code and hyperparameter tuning, leading in the end to better models. In summary, researchers should scale their training with the usage of *Horovod* or *PyTorch-DDP* to as much GPUs as possible, until the degradation of the accuracy sets in.

There exist other distributed DL frameworks that could be analyzed in terms of performance on large HPC systems. Furthermore, on the hardware side, different processor architectures that are tailored for machine learning, e.g., TPUs or Graphocore IPUs [49], are emerging and will undoubtedly play a key role in distributed DL. In the future, these developments will be further investigated.

The issue with the accuracy drop for large batch sizes also requires further attention. Other promising techniques apart from learning rate scheduling methods include novel optimizers, e.g., NVLAMB [50]. A comprehensive hyperparameter tuning routine, which includes other optimizer-related parameters, such as weight-decay rate or momentum can also impact the performance.

While this work evaluated the data loaders and frameworks already at large scale, further scaling tests are needed for even bigger systems, e.g., Exascale machines. Therefore, larger datasets and more complex models will be required. Other model architectures such as Autoencoders or Transformers have shown great success on various tasks and hence might be a good choice.

Future directions of general big data research include developing more efficient and adaptive distributed DL algorithms to handle heterogeneous data sources, reduce storage and communication overheads in HPC systems, and perform energy efficiency training of DL models on massive datasets.

Abbreviations

API	Application Programming Interface
BS	Batch Size
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DALI	NVIDIA Data Loading Library
DL	Deep Learning
DRAM	Dynamic Random Access Memory
GPU	Graphics Processing Unit
HPC	High-Performance Computing
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
I/O	Input/Output
IPU	Intelligent Processing Unit
JUWELS	Jülich Wizard for European Leadership Science (JUWELS)
LR	Learning Rate
MPI	Message Passing Interface
MSA	Modular Supercomputing Architecture
NCCL	NVIDIA Collective Communications Library
PCIe	Peripheral Component Inter-Connect Express
ResNet	Residual Neural Network
SGD	Stochastic Gradient Descent
TPU	Tensor Processing Unit
V	Validation Accuracy

Acknowledgements
Not applicable.

Author contributions
MA: writing code and paper, running experiments and analysis, EI: contributions to code and analysis, review, RS: contributions to code and analysis, review, MR: supervision, contribution to analysis, review, AL: supervision, funding, contribution to analysis, review. All authors read and approved the final manuscript.

Funding
Open Access funding enabled and organized by Projekt DEAL. The research leading to these results has been conducted in the CoE RAISE project, which receives funding from the European Union's Horizon 2020-Research and Innovation Framework Programme H2020-INFRAEDI-2019-1 under Grant Agreement No. 951733. The authors gratefully acknowledge the Partnership for Advanced Computing in Europe (PRACE) for funding this project by providing computing time on the Supercomputer JUWELS [8] at Jülich Supercomputing Centre (JSC). Open Access publication funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 491111487.

Availability of data and materials
The code to reproduce the experiments is available to the public via https://gitlab.jsc.fz-juelich.de/CoE-RAISE/FZJ/resnet-benchmarks
The datasets generated and/or analysed during the current study are available in the ImageNet repository, https://www.image-net.org/download.php.

Declarations

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Received: 3 November 2022 Accepted: 8 May 2023
Published online: 08 June 2023

References
1. He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2016;pp. 770–778. https://doi.org/10.1109/CVPR.2016.90
2. Dosovitskiy A, Beyer L, Kolesnikov A, Weissenborn D, Zhai X, Unterthiner T, Dehghani M, Minderer M, Heigold G, Gelly S, Uszkoreit J, Houlsby N. An image is worth 16x16 words: Transformers for image recognition at scale. In: International Conference on Learning Representations (2021). arxiv:2010.11929
3. Ben-Nun T, Hoefler T. Demystifying parallel and distributed deep learning: an in-depth concurrency analysis. ACM Comput Surv. 2019. https://doi.org/10.1145/3320060.
4. Goyal P, Dollár P, Girshick R, Noordhuis P, Wesolowski L, Kyrola A, Tulloch A, Jia Y, He K. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour 2018. arXiv:1706.02677
5. Sergeev A, Balso M.D. Horovod: fast and easy distributed deep learning in TensorFlow 2018. arXiv:1802.05799
6. Li S, Zhao Y, Varma R, Salpekar O, Noordhuis P, Li T, Paszke A, Smith J, Vaughan B, Damania P, Chintala S. PyTorch distributed: experiences on accelerating data parallel training. Proc VLDB Endow. 2020;13(12):3005–18. https://doi.org/10.14778/3415478.3415530.
7. Rasley J, Rajbhandari S, Ruwase O, He Y. DeepSpeed: system optimizations enable training deep learning models with over 100 billion Parameters, pp. 3505–3506. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3394486.3406703
8. Jülich Supercomputing Centre. JUWELS: Modular Tier-0/1 Supercomputer at Jülich Supercomputing Centre. J Large Scale Res facil JLSRF. 2019;5:135. https://doi.org/10.17815/jlsrf-5-171
9. Top500. https://top500.org/lists/top500/list/2022/06/. Accessed: 2022-09-20
10. Russakovsky O, Deng J, Su H, Krause J, Satheesh S, Ma S, Huang Z, Karpathy A, Khosla A, Bernstein M, Berg AC, Fei-Fei L. ImageNet large scale visual recognition challenge. Int J Comput Vision (IJCV). 2015;115(3):211–52. https://doi.org/10.1007/s11263-015-0816-y.
11. Mattson P, Cheng C, Diamos G, Coleman C, Micikevicius P, Patterson D, Tang H, Wei G-Y, Bailis P, Bittorf V, Brooks D, Chen D, Dutta D, Gupta U, Hazelwood K, Hock A, Huang X, Kang D, Kanter D, Kumar N, Liao J, Narayanan D, Oguntebi T, Pekhimenko G, Pentecost L, Janapa Reddi V, Robie T, St John T, Wu C-J, Xu L, Young C. Zaharia, M. Mlperf training benchmark. In: Dhillon, I., Papailiopoulos, D., Sze, V. (eds.) Proceedings of Machine Learning and Systems. 2020;vol 2:336–49.
12. Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. Commun ACM. 2017;60(6):84–90. https://doi.org/10.1145/3065386.
13. Ioffe S, Szegedy C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: Bach F, Blei D. (eds.) Proceedings of the 32nd International Conference on Machine Learning. Proceedings of Machine Learning Research 2015: vol. 37, pp. 448–456. PMLR, Lille, France.
14. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Lu, Polosukhin I. Attention is all you need. Advances in Neural Information Processing Systems. 2017;30.
15. Li D, Chen X, Becchi M, Zong Z. Evaluating the energy efficiency of deep convolutional neural networks on cpus and gpus. In: 2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom), pp. 477–484 2016. https://doi.org/10.1109/BDCloud-SocialCom-SustainCom.2016.76.

16. Strubell E, Ganesh A, McCallum A. Energy and policy considerations for deep learning in NLP. In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, pp. 3645–3650. Association for Computational Linguistics, Florence, Italy 2019. https://doi.org/10.18653/v1/P19-1355
17. Langer M, He Z, Rahayu W, Xue Y. Distributed training of deep learning models: a taxonomic perspective. IEEE Trans Parallel Distributed Syst. 2020;31(12):2802–18. https://doi.org/10.1109/tpds.2020.3003307.
18. Dean J, Corrado G, Monga R, Chen K, Devin M, Mao M, Ranzato, MA, Senior A, Tucker P, Yang K, Le Q, Ng A. Large scale distributed deep networks. Advances in Neural Information Processing Systems. 2012;25.
19. Xing EP, Ho Q, Dai W, Kim JK, Wei J, Lee S, Zheng X, Xie P, Kumar A, Yu Y. Petuum: a new platform for distributed machine learning on big data. IEEE Trans Big Data. 2015;1(2):49–67. https://doi.org/10.1109/TBDATA.2015.2472014.
20. Chilimbi T, Suzue Y, Apacible J, Kalyanaraman K. Project Adam: building an efficient and scalable deep learning training system. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. OSDI'14, 2014: pp. 571–582. USENIX Association, USA.
21. Iandola FN, Moskewicz M, Ashraf K, Keutzer K. FireCaffe: near-linear acceleration of deep neural network training on compute clusters. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 2592–2600 (2016). arxiv:1511.00175
22. Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, et al. MD. TensorFlow: large-scale machine learning on heterogeneous systems (2015). arxiv:1603.04467
23. Chen T, Li M, Li Y, Lin M, Wang N, Wang M, Xiao T, Xu B, Zhang C, Zhang Z. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems 2015. arxiv:1512.01274
24. Götz M, Debus C, Coquelin D, Krajsek K, Comito C, Knechtges P, Hagemeier B, Tarnawa M, Hanselmann S, Siggel M, et al. Heat—a distributed and gpu-accelerated tensor framework for data analytics. 2020 IEEE International Conference on Big Data (Big Data) 2020. https://doi.org/10.1109/bigdata50022.2020.9378050.
25. Numpy. https://numpy.org/. Accessed 20 Sep 2022.
26. Rajbhandari S, Rasley J, Ruwase O, He Y. Zero: Memory optimizations toward training trillion parameter models. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–16 (2020). https://doi.org/10.1109/SC41405.2020.00024
27. Shams S, Platania R, Lee K, Park S-J. Evaluation of deep learning frameworks over different hpc architectures. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), pp. 1389–1396 (2017). https://doi.org/10.1109/ICDCS.2017.259
28. SLURM. https://slurm.schedmd.com/. Accessed 20 Sep 2022.
29. Yamazaki M, Kasagi A, Tabuchi A, Honda T, Miwa M, Fukumoto N, Tabaru T, Ike A, Nakashima K. Yet another accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds 2019. arxiv:1903.12650.
30. Kumar S, Bradbury J, Young C, Wang YE, Levskaya A, Hechtman B, Chen D, Lee H, Deveci M, Kumar N, Kanwar P, Wang S, Wanderman-Milne S, Lacy S, Wang T, Oguntebi T, Zu Y, Xu Y, Swing A. Exploring the limits of concurrency in ML training on Google TPUs. 2021. arxiv:2011.03641
31. Krizhevsky A. One weird trick for parallelizing convolutional neural networks. 2014. arxiv:1404.5997
32. Szegedy C, Vanhoucke V, Ioffe S, Shlens J, Wojna Z.D Rethinking the inception architecture for computer vision. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 2818–2826 (2016). https://doi.org/10.1109/CVPR.2016.308.
33. Pumma S, Buono D, Checconi F, Que X, Feng W-C. Alleviating load imbalance in data processing for large-scale deep learning. In: 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), pp. 262–271 (2020). https://doi.org/10.1109/CCGrid49817.2020.00-67.
34. Gibiansky A. Bringing HPC techniques to deep learning 2017. https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/. Accessed 31 Aug 2021.
35. NCCL. https://developer.nvidia.com/nccl. Accessed 20 Sep 2022.
36. Gloo. https://github.com/facebookincubator/gloo. Accessed 20 Sep 2022.
37. DALI. https://developer.nvidia.com/dali. Accessed 20 Sep 2022.
38. Zolnouri M, Li X, Nia V.P. Importance of data loading pipeline in training deep neural networks 2020. arxiv:2005.02130.
39. Wang G, Lei Y, Zhang Z, Peng C. A communication efficient ADMM-based distributed algorithm using two-dimensional torus grouping AllReduce. Data Sci Eng. 2023;1–12.
40. Zhou Q, Kousha P, Anthony Q, Shafie Khorassani K, Shafi A, Subramoni H, Panda DK. Accelerating MPI all-to-all communication with online compression on modern GPU clusters. In: High Performance Computing: 37th International Conference. ISC High Performance 2022. Hamburg, Germany: Springer; 2022. p. 3–25.
41. Bengio Y, Simard P, Frasconi P. Learning long-term dependencies with gradient descent is difficult. IEEE Trans Neural Netw. 1994;5(2):157–66. https://doi.org/10.1109/72.279181.
42. Loshchilov I, Hutter F. SGDR: Stochastic gradient descent with warm restarts. International Conference on Learning Representations (2017).
43. Suarez E, Eicker N, Lippert T. Modular supercomputing architecture: from idea to production. Contemporary high performance computing. 2019;23–55. https://doi.org/10.1201/9781351036863-9.
44. NVLINK. https://www.nvidia.com/en-us/data-center/nvlink/. Accessed 20 Sep 2022.
45. GPFS. https://apps.fz-juelich.de/jsc/hps/juwels/filesystems.html. Accessed 17 Apr 2023.
46. EasyBuild. https://github.com/easybuilders/easybuild. Accessed 20 Sep 2022.
47. NSys. https://docs.nvidia.com/nsight-systems/index.html. Accessed 20 Sep 2022.
48. cuDNN. https://developer.nvidia.com/cudnn. Accessed 20 Sep 2022.
49. Graphcore. https://www.graphcore.ai/products/ipu. Accessed 20 Sep 2022.
50. NVLAMB. https://github.com/NVIDIA/DeepLearningExamples/blob/master/PyTorch/LanguageModeling/BERT/README.md. Accessed 20 Sep 2022.