Journal of Big Data

RESEARCH Open Access

Accelerating neural network training with distributed asynchronous and selective optimization (DASO)

Daniel Coquelin[1*], Charlotte Debus[1], Markus Götz[1], Fabrice von der Lehr[2], James Kahn[1], Martin Siggel[2] and Achim Streit[1]

*Correspondence: daniel.coquelin@kit.edu

[1] Karlsruhe Institute of Technology, Hermann-von-Helmholtz-Platz 1, 76344 Eggenstein-Leopoldshafen, Germany
Full list of author information is available at the end of the article

Abstract

With increasing data and model complexities, the time required to train neural networks has become prohibitively large. To address the exponential rise in training time, users are turning to data parallel neural networks (DPNN) and large-scale distributed resources on computer clusters. Current DPNN approaches implement the network parameter updates by synchronizing and averaging gradients across all processes with blocking communication operations after each forward-backward pass. This synchronization is the central algorithmic bottleneck. We introduce the distributed asynchronous and selective optimization (DASO) method, which leverages multi-GPU compute node architectures to accelerate network training while maintaining accuracy. DASO uses a hierarchical and asynchronous communication scheme comprised of node-local and global networks while adjusting the global synchronization rate during the learning process. We show that DASO yields a reduction in training time of up to 34% on classical and state-of-the-art networks, as compared to current optimized data parallel training methods.

Keywords: Machine learning, Neural networks, Data parallel training, Multi-node, Multi-GPU, Stale gradients

Introduction

Recent advances in deep learning have thrived under the theme “bigger is better”. Modern neural networks yield super-human performance on problems such as image classification and semantic segmentation by introducing higher model complexity [1, 2]. However, the training of large networks requires large datasets. As the sizes of models and datasets increase, so do the computational resources required. Put simply, today’s deep learning tasks are limited by the hardware and computing time available. In response, parallel training methods have been developed to enable the concurrent use of multiple (distributed) hardware devices.

In general, there are three approaches to parallel training [3]: model parallelism, pipelining, and data parallelism. The model parallel approach distributes the network across multiple computing devices, for example two GPUs with half of the network each [4–6]. Pipelining is a special case of model parallelism [7–9]. In the context of neural networks it refers to placing entire model components, e.g. layers, on different devices then sending the results from one device to the next sequentially. In the data parallel approach, each available computing device trains an identical copy of the network and synchronizes its model state with the other devices. [10–12]

Data parallel neural networks (DPNNs) have been used on various architectures and data types to train state-of-the-art models in a fraction of the time required to train a model traditionally [13, 14]. Each model instance in a DPNN performs a forward-backward pass over a unique and disjoint portion of the data, called a shard, after which the parameters of all networks are synchronized using a global collective operation. This can be effectively viewed as one batch distributed across the devices, i.e. a distributed batch. Typically, the synchronization of network parameters is a blocking averaging operation [3]. This collective blocking operation comprises an inherent bottleneck as it waits for all models to exchange network parameters or gradients before further calculations can occur. Using non-blocking communication operations can provide some relief as the next forward-backward step can begin while communication is ongoing. However, as global parameter updates are running asynchronously, parameters found by individual network instances are always slightly out-of-date, or stale.

Although computing devices can take many forms, GPUs are currently the most efficient and powerful for training neural networks. Therefore, we will refer to computing devices as GPUs throughout this paper. Commonly, the standard communication structure communicates with GPUs individually to synchronize network parameters. This neglects the structure of most computer clusters, where multiple GPUs are grouped on computing nodes with significantly faster node-local connections as compared to inter-node communication. Large multi-node DPNNs can instead be divided into node-local DPNNs which are themselves members of a global DPNN. This hierarchical approach could significantly reduce the communication overhead, as less data is sent between nodes. Furthermore, what if global parameter synchronization did not occur after every batch and instead, the average was calculated asynchronously every B^{th} batch? This would help to further alleviate the aforementioned communication bottleneck and could greatly accelerate training.

We present our contributions in the follow list, ordered by importance.

- The distributed selected and asynchronous optimization (DASO) method.
- A parameter study of DASO's primary parameters with detailed analysis.
- A performance evaluation of DASO as compared to Horovod [15] and a classic synchronous DPNN training method on two different tasks.
- An outlook on the future research that can be done on DASO.

DASO performs communication for network parameter updates in a hierarchical manner: on the node-local level, in the form of GPU-to-GPU communication operations, and on the global level, where computing nodes are treated as individual entities. This approach allows DASO to perform the time-expensive global synchronization step asynchronously, with stale gradients, and after multiple batches instead of after every forward-backward pass, thus leveraging the potential of acceleration via parallel computation on modern computer clusters.

The remainder of this paper is organized as follows. In "Related work" section we will discuss relevant work previously done in the area of data parallel model training. "Proposed work: distributed asynchronous and selective optimization (DASO)" section introduces the concept of distributed asynchronous and selective optimization, followed by a parameter study and two performance evaluations on the tasks of image classification and semantic segmentation in "Results and discussion" section. Our results are summarized and discussed in "Conclusions and future works" section, which also gives an outlook towards further improvement and application of the method.

Related work

Data parallelism is the go-to option for accelerating neural network training on large datasets. In DPNNs, each local network is optimized locally, e.g using mini-batch stochastic gradient descent (SGD) [16], before the optimization results are synchronized with all other networks. The most straightforward approach to global synchronization is a collective blocking, average operation after every forward-backward step. This inherently limits the speed of the data parallel training.

Recently, advancements have been made in accelerating the synchronization process by starting the communication of gradient updates while the backward pass is ongoing, with one reporting training times of only 74.7s on the ImageNet data set [13]. This approach has been shown to be quite effective, but the process of tuning the communication patterns does not generalize well, as it is highly dependent on the specific neural network architecture.

Several works have investigated the use of asynchronous SGD (ASGD) [17–19], which updates the parameters whenever a network finishes a backward pass. Each network retrieves the current model parameters from a parameter server before performing a forward-backward pass. After finishing the backward step, the network sends its updated parameters to the server, which determines the new global parameters using the updates from all processes. However, if a network is still computing the forward-backward pass when the parameter server is updated, the network's current parameters become stale.

Stale gradients can be leveraged to approximate accurate network parameters in a variety of SGD variants [20], and ASGD has been shown to yield consistent convergence [21]. Recent attempts at further accelerating ASGD have been made using individual network optimizers for a warm-up phase and delayed updates to the parameter server [22].

Hierarchical algorithms are a typical approach for maximizing the usage of computing clusters. This approach has been used to accelerate synchronous SGD with positive results. Local SGD, post-local SGD, and hierarchical SGD [23] propose methods of local and global update steps, each occurring after a fixed number of forward-backward passes.

PyTorch [24] and TensorFlow [25] are currently the most widely used machine learning frameworks. Both offer options for traditional data parallel training. For large systems, a global communication protocol, such as MPI [26], is often required to leverage specialized inter-node connections. Recently, there have been many advancements in the optimization of the global parameter synchronization operation by using MPI with multiple network topologies [27, 28]. These approaches have shown promising results, but remain centered around the idea of a global synchronization for each forward-backward pass.

Currently, the most popular MPI-enabled DPNN framework is Horovod [15]. To reduce the size of data sent via the communication network, Horovod uses tensor fusion, or grouping parameters together to be communicated in a larger chunk of data, and data compression. Using the grouped parameters, communication can be started during the backward pass and the data within the buckets can be received during the next forward pass. The data compression in Horovod is frequently done by quantizing the network parameters into 16-bit floating-point format.

Proposed work: distributed asynchronous and selective optimization (DASO)

The common approach to training DPNNs is to perform a forward-backward pass on each network instance with one portion of the distributed batch, then synchronize the network parameters via a global averaging operation. The averaging of gradients is only an approximation of the true gradients that would be calculated for the entire dataset. This approximation is made under the assumption that each portion of the distributed batch is independent and identically distributed (iid) [29], i.e. the disjoint subsets are representative of the dataset as a whole. With this assumption, the traditional update function for distributed SGD is

$$x_{t+1} = x_t - \eta \frac{\sum_{i=1}^{P} x_i}{P} \tag{1}$$

where x_{t+1} is the model state for batch $t+1$, η is the learning rate, and P is the number of processes.

Under the iid assumption, another approximation can be made: the average parameters of a subset of networks are not significantly different than the average parameters of the complete set of networks. Recalling that modern cloud systems, clusters, and supercomputers have different inter- and intra-node communication capabilities (with different bandwidths and latencies), we can utilize this approximation to reduce the communication needed for parallel training, thereby alleviating the intrinsic bottleneck of blocking synchronizations.

Based on these foundational concepts we propose the Distributed Asynchronous and Selective Optimization (DASO) method. Instead of a uniform communications network across multiple multi-GPU nodes, DASO employs a hierarchical network model with node-local networks and a global network.

The global network spans all GPUs on all nodes, while the node-local networks are composed of the GPUs on each individual node. The global network is divided into

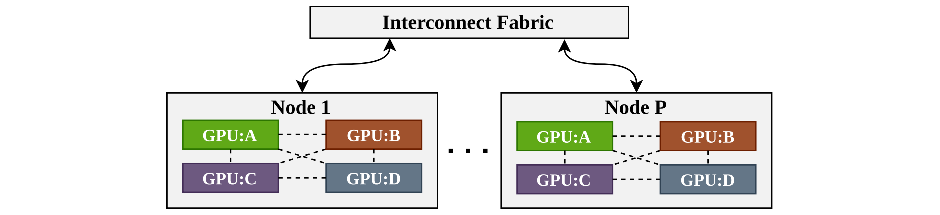


Fig. 1 MPI Groups An overview of a common node-based computer cluster with P nodes and four GPUs per node. GPU colors represent *group* membership. The dashed lines indicate GPU-to-GPU communication channels

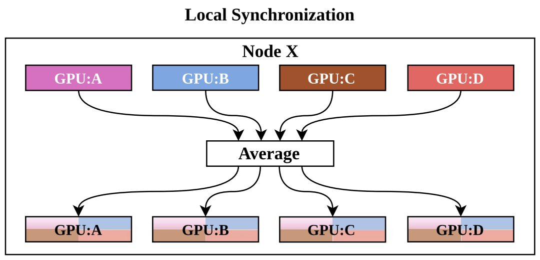


Fig. 2 Local synchronization Schematic of the local synchronization step for a single node with four GPUs. The gradients from each GPU are averaged, then each GPU's gradients are set to the result

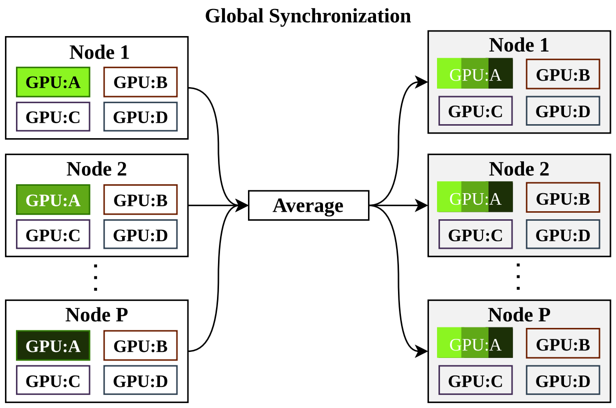


Fig. 3 Global synchronization Schematic of the global synchronization step performed by the global communication *group* consisting of GPU:A on each node. The network parameters are averaged by each GPU in the *group*, and the network parameters of each *group* member are set to the result

multiple communication *groups*, with each *group* containing a single GPU from every node. Global communication takes place exclusively within a *group*, i.e. only *group* members exchange data, while members of other *groups* do not participate. Communication between the node-local GPUs is then handled by the local network, which benefits from high-speed GPU-to-GPU interconnects and optimized communication packages (e.g. NCCL [30]). Under the assumption that the cluster node configurations are homogeneous, DASO creates *groups* between GPUs with the same node-local identifier as is

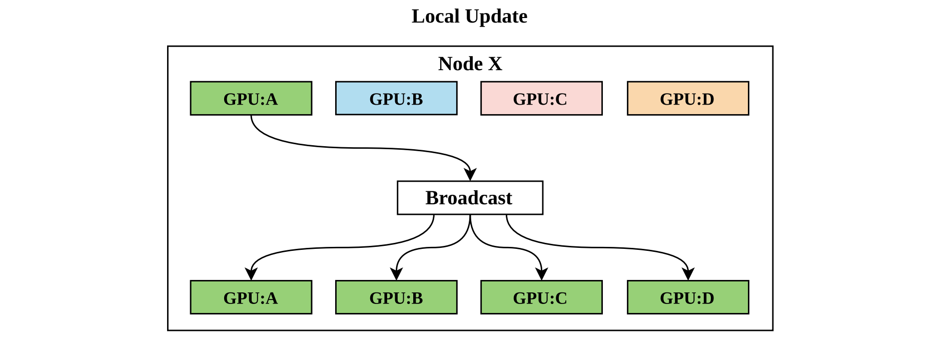


Fig. 4 Local Update (Broadcast) Schematic of the local update step to be performed after the global synchronization step shown in Fig. 3. The group member responsible for the global communication, in this case GPU:A, sends its network parameters to all other node-local GPUs, which replace the old parameters on those GPUs

shown in Fig. 1. With this approach, inter-node communication can be reduced by a factor equal to the minimum number of GPUs per node.

Similarly to local-SGD [23], DASO utilizes a multi-step synchronization. Local synchronization (Fig. 2) occurs after each batch and uses the node-local network to do gradient-averaging between the local GPUs. The local synchronization is done using Eq. (1) where P is the number of GPUs on a single node. Global synchronization (Fig. 3) occurs after one or more local synchronizations, in which the network parameters of all members of a single global *group* are shared and averaged. Following every global synchronization, a local update function broadcasts the averaged parameters from the local *group* member to all other node-local GPUs (Fig. 4). The role of global synchronization rotates between *groups* to balance the communications load.

Global synchronization can be performed in a blocking or non-blocking manner. In the blocking case, all synchronization steps are performed after each batch. To reduce the amount of data transferred, parameters are cast to a 16-bit datatype representation during buffer packaging. This operation does not greatly affect convergence, as shown by Alistarh et al. [31]. Once received, the parameters are cast back to their original datatype. In the non-blocking case, the next forward-backward pass is started after the parameters are sent but before they are received. Datatype casting is not beneficial in this scenario, as it delays the start of parameter communications.

After the parameters are sent during a global synchronization, each neural network conducts B forward-backward passes with local synchronization before the *group* members receive the sent parameters. Hence, the updates from the global communication step are stale upon their arrival. To compensate for this, a weighted average of the stale global parameters and the current local parameters is calculated as follows:

$$x_{t+S} = \frac{2Sx^{l}_{t+S-1} + \sum_{i=1}^{P} x^{i}_{t}}{2S + P} \tag{2}$$

where x^{l}_{t+S} is the model state on GPU l, S is the number of batches after batch t; x^{i}_{t} is the model state of GPU i after batch t; and P is the number of GPUs in the global network.

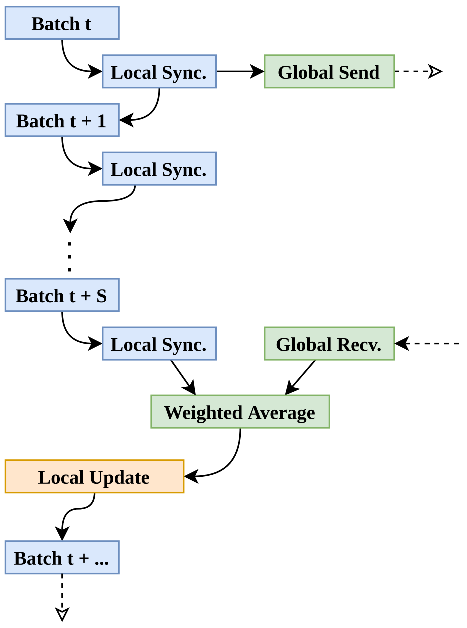


Fig. 5 Cycling Flow Process flow diagram of the synchronization steps during the cycling phase where *t* is the batch number and *S* is the batches to wait before global synchronization. The weighted average is calculated as shown in Eq. (2)

The weighting of the local parameters was found experimentally. A detailed explanation of Eq. (2) and its validity is provided in Additional file 1 ("Conclusions and future works" section).

Training of a network with the DASO method can be divided into three phases:

1. Warm-up
2. Cycling
3. Cool-down

The warm-up and cool-down phases utilize blocking global synchronizations, while the cycling phase uses non-blocking global synchronizations. Given a fixed number of total epochs, the warm-up and cool-down phases occur for a set number of epochs at the beginning and end of training, respectively. The warm-up phase is used to quickly move away from the randomly initialized parameters and prepare for the cycling phase. The cool-down phase is intended to refine the network parameters at the end of training.

In the cycling phase, the number of forward-backward passes between global synchronizations (B) and the number of batches to wait for global synchronization data (S) are varied. B is specified manually upon initialization. When the training loss plateaus, B and S are reduced by a factor of two, down to a minimum of one. When $B, S = 1$ and the loss has plateaued, both are reset to their initial values and the process is repeated until the cool-down phase. The synchronization steps in the cycling phase are schematically shown in Fig. 5. The phasic structure of DASO is intended to maintain the network accuracy as best as possible, while reducing the training time.

The synchronization steps in the cycling phase can also be described in a list starting from a given batch t:

1. Perform local synchronization on all nodes.
2. Members of a single MPI group send their parameters to each other in a non-blocking fashion. This is the first step in the global synchronization.
3. S batches are performed using *only* node-local synchronization.
4. After batch $t + S$, receive the parameters send in step 2.
5. Compute the weighted average to update the network parameters.

Implementation

A DASO proof-of-concept is currently implemented in the Heat framework [32] for usage with PyTorch networks. Heat is an open-source Python framework for distributed and GPU-accelerated data analytics, which offers both low level array computations as well as assorted higher-level machine learning algorithms. The local networks utilize PyTorch's `DistributedDataParallel` class and distributed package [33]. The global communication network utilizes Heat's MPI backend, which handles the automatic communication of PyTorch tensors. The global *groups* are implemented as MPI groups.

To use this implementation of DASO to train an existing PyTorch network, only four additional functions need to be called and the data loaders need to be modified to distribute the data between all GPUs.[1] The function calls are illustrated in Listing 1. First, the node-local PyTorch processes are created, which will be utilized during the local synchronization step. Next, the DASO instance is created with a PyTorch node-local optimizer (e.g. SGD) and the number of epochs for training is specified. The DASO instance will find the aforementioned PyTorch processes automatically.

Table 1 Hyperparameters used to train ResNet-50 using the ImageNet-2012 dataset

Data Loader	**DALI [37]**	
Local Optimizer	SGD	
Local Optimizer Parameters	Momentum: 0.9	Weight Decay: 0.0001
Epochs	90	
Learning Rate (LR) Decay	Reduce on Stable	
LR Parameters	Stable Epochs Before Change: 5	Decay Factor: 0.5
LR Warmup Phase	5 epochs, see Goyal et al. [38]	
Maximum LR	Scaled by number of GPUs [38]	
Loss Function	Cross Entropy	

[1] The data loaders need only know how many GPUs exist and what their global rank, i.e. ascending integral ID, is.

```
import heat as ht
import torch
...
# create PyTorch distributed group
world_size = ht.MPI_WORLD.size
rank = ht.MPI_WORLD.rank
local_rank = rank % num_local_gpus
torch.distributed.init_process_group(
    backend="nccl",
    rank=local_rank,
    world_size=world_size
)
...
# the DASO optimizer is created
daso_optimizer = ht.optim.DASO(
    local_optimizer=optimizer,
    total_epochs=num_epochs
```

[1]The data loaders need only know how many GPUs exist and what their global rank, i.e. ascending integral ID, is.

```
)
...
# the hierarchical network is created
ht_model = ht.nn.DataParallelMultiGPU(
    net,
    daso_optimizer
)
```

Results and discussion

All experiments were conducted on the JUWELS Booster at the Jülich Supercomputing Center [34]. This high-performance computing cluster has 936 GPU nodes, each with two AMD EPYC Rome CPUs and four NVIDIA A100 GPUs [35], connected via an NVIDIA Mellanox HDR InfiniBand interconnect fabric. The following software versions were used: CUDA 11.0, ParaStationMPI 5.4.7-1-mt, Python 3.8.5, PyTorch 1.7.1+cu110, Horovod 0.21.1, and NCCL 2.8.3-1. The JUWELS Booster provides a CUDA-aware MPI implementation, meaning that GPUs can communicate directly with other GPUs.

Parameter study

As previously stated, stale parameters can effect both the accuracy and training time of a network. However, the effects of stale global updates when combined with node-local synchronous data parallel training are not known. Therefore, it is important to determine how frequently global synchronizations must occur, and how staleness affects both accuracy and speed. To this end, we performed a parameter study using the ImageNet-2012 dataset [36] to train a ResNet-50 [1] neural network using either 32 or 128 GPUs with fixed numbers of batches between global synchronizations, B, and S batches between the sending and receiving of the global parameters. B and S do not change for the entirety of each measurement, i.e. the training phases described above are disabled.

ImageNet-2012 is a large dataset containing 1.2 million labeled images. We evaluate classification quality using top-1 accuracy, i.e. the accuracy with which the model predicts the image labels correctly with a single attempt. File loading from disk and pre-processing steps utilized DALI [37]. Training hyperparameters are shown in Table 1.

Table 2 Parameter study results. *B* is the number of forward-backward passes between global synchronizations and *W* is the number of batches to wait for the global synchronization data

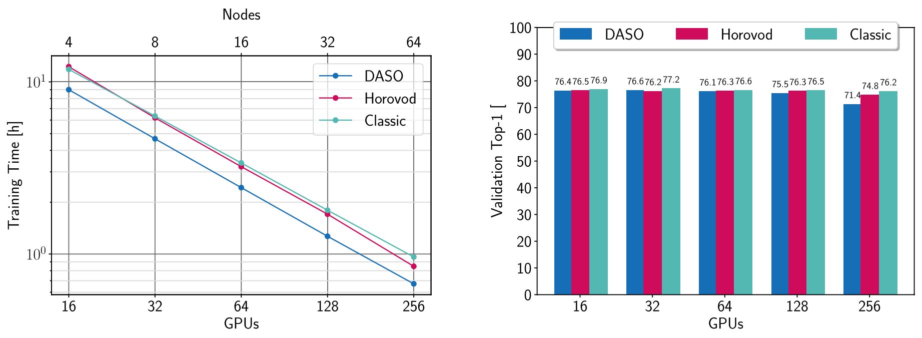
		32 GPUs (8 nodes)		128 GPUs (32 nodes)	
B	***S***	**Runtime, h**	**Validation Top-1, %**	**Runtime, h**	**Validation Top-1, %**
1	0	4.5606	76.7715	1.2064	76.5416
1	1	4.2545	76.0859	1.1556	74.9233
2	0	4.0365	76.8828	1.0769	76.3027
2	1	3.8943	75.8086	1.0427	74.8936
2	2	3.8919	75.9238	1.0450	75.0854
4	0	3.6984	76.4258	0.9775	74.3478
4	1	3.6560	75.8262	1.0142	73.2962
4	2	3.7191	75.5020	0.9843	71.9570
4	4	3.7064	75.7070	0.9784	73.8560
8	0	3.4922	75.2598	0.9078	69.2732
8	4	3.5259	74.6113	0.9170	65.4733
8	8	3.5770	75.2637	0.9302	69.6655
16	0	3.3235	73.1348	0.8585	58.5397
16	4	3.3417	73.1758	0.8590	56.8865
16	8	3.3934	73.2148	0.8724	54.5323
16	16	3.4828	74.2129	0.8933	62.3692
32	0	3.2224	70.7480	0.8231	43.6855
32	4	3.2302	70.2773	0.8247	44.0639
32	16	3.2969	69.5781	0.8430	41.2458
32	32	3.4083	72.5488	0.8656	50.9539

They are adapted from the example script in PyTorch for training ResNet-50 on the ImageNet dataset and the work done by Goyal et al. [38].

The results of these experiments are shown in Table 2.

The effects of skipping global synchronizations can be seen most clearly for the measurements when S was set to zero, in which case the optimizer does not use stale gradients. The anti-correlation between network accuracy and B is apparent for both node configurations, but more pronounced for the 128 GPU measurements. As the expected accuracy of this network is around 76%, these measurements show that there is negligible loss of accuracy, while significantly reducing training time, when B is less than or equal to four. Furthermore, as B increases, the time required to train the network decreases at the cost of classification accuracy.

Stale gradients are known to negatively effect the accuracy of a network unless they are handled specifically [21]. A partial conformation of this is shown for the measurements when B is held constant. As expected, the accuracy begins to decrease as S increases. However, when B is equal to S, the accuracy improves again. In some cases, it improves to a point of higher accuracy than reached with the $S = 0$ measurement. Detailed studies are required to further investigate this effect. Our results indicate that the stale gradients may have regularizing effects.



(a) **Training Time** (b) **Validation Top1**

Fig. 6 ImageNet ResNet-50 training times and top-1 accuracy results on the ImageNet dataset when trained with DASO, Horovod, and the classic algorithm for increasing node counts. Each node has four GPUs

Performance evaluation

We evaluated DASO's computational performance on two common examples of data-intensive neural network challenges: (1) image classification and (2) semantic segmentation For image classification, we trained a ResNet-50 [1] on the ImageNet-2012 [36] dataset. This can be considered a standard benchmark for machine learning, since pre-trained ResNet-50 networks are the backbone of many computer vision pipelines [39]. For semantic segmentation, we trained a state-of-the-art hierarchical multi-scale attention network [14] on the Cityscapes [40] dataset.

We compared DASO with Horovd and a classical naïve data parallel optimization method. The classic method does not use compression or tensor fusion, but it does begin the communication during the backward step and receives the data during the forward step. To achieve better comparability, this approach was also implemented in Heat alongside DASO. Horovod generally uses a strategy similar to the classical approach. However, it additionally utilizes compression techniques, so-called tensor fusions, as well as other optimizations to accelerate training (e.g. the use of threads with MPI). Horovod is currently the most popular choice for data parallel training of neural networks on computer clusters. We elected not to compare with PyTorch's distributed package as it utilizes a similar approach to Horovod, namely compression and bucketing. The performance evaluation compares the strong scaling behavior of all approaches with respect to training time and the task specific target metric.

The networks' hyperparameters remain constant for all experiments. All tested networks use a learning rate scheduler. When the training loss plateaus, i.e. the training loss is not decreasing by more than a set percentage threshold, the scheduler decreases the learning rate by a set factor. Settings of the scheduler, as well as for the local optimizer settings, were set to be identical for all optimizers for each use-case. With respect to message packaging, Horovod was configured to use floating-point 16 bit compression, DASO compresses to brain floating-point 16. As the classic method sends each set of parameters individually, compression and decompression results in an increase in the training time. Therefore, the classic method uses floating-point 32 bit for communication operations. Compression to brain floating-point 16 for communication is not currently available in Horovod, but does not effect communication bandwidth or latency.

Table 3 Hyperparameters used to train the hierarchical multi-scale attention network using the Cityscapes dataset

Data Loader	PyTorch	
Local Optimizer	SGD	
Local Optimizer Parameters	Momentum: 0.9	Weight Decay: 0.0001
Epochs	175	
Learning Rate (LR) Decay	Reduce on Stable	
LR Parameters	Stable Epochs Before Change: 5	Decay Factor: 0.75
LR Warmup Phase	5 epochs, see Goyal et al. [38]	
Maximum LR	0.4	
Loss Function	Region Mutual Information [42]	

The training batch size is fixed for each GPU in all experiments. Hence, the combined distributed batch size increases by the number of GPUs times the local batch size.

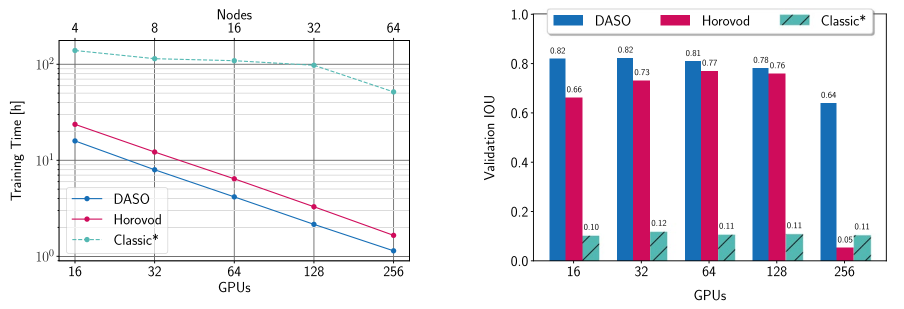
Utilizing the results shown in "Parameter study" section, DASO's maximum number of batches between global synchronizations was set to four and the number of batches between sending and received the global parameters was set to one for these experiments. These values were chosen with the goal of balancing speed and accuracy as the number of training devices was increased.

Image classification—ImageNet

This experiment was conducted using the ResNet-50 architecture on the ImageNet-2012 dataset [36]. The network training configuration is the same as those mentioned in "Parameter study" section . DASO's settings are those which have been stated in "Proposed work: distributed asynchronous and selective optimization (DASO)" section. The network hyperparameters can be found in Table 1.

Training was conducted on four, eight, 16, 32, and 64 nodes, which equates to 16, 32, 64, 128, and 256 GPUs, respectively. This roughly corresponds to traditional strong scaling experiments for parallel algorithms, where an exponential increase in nodes should ideally result in a proportional reduction in time, given a constant computational load. The results of the experiment are shown in Fig. 6a. DASO, Horovod, and the classic algorithm show desirable strong scaling behavior, i.e. a factor of two in GPU number results in the training time being roughly halved. However, the scaling of the classic method begins to worsen as the number of GPUs increases. Due to DASO's optimized hierarchical communication scheme and the reduced number of synchronizations, DASO requires up to 25% less training time than Horovod for this task.

Up to 128 GPUs, DASO and Horovod yield similar levels of accuracy, while the classic method outperforms both, see Fig. 6b. With more than 128 GPUs, DASO and Horovod did not exceed 75% top-1 accuracy and the classic algorithm appears to be unchanged. The drop off can be in part explained by the fact that accuracy starts to a decrease at larger batch sizes in a traditional network, unless special allowances are made [38]. Since we keep the portion of the distributed batch that is processed on each individual GPU the same, larger GPU counts ultimately result in a larger distributed batch. Hence,



(a) **Training Time** As the classical network hit the time limit, the values are estimated.

(b) **Maximum IOU** Classic network accuracy values are the best results when training was stopped.

Fig. 7 Cityscapes Benchmarking results for the selected hierarchical split level attention network [14] on the Cityscapes dataset with DASO, Horovod, and the classic DPNN method for increasing node counts, each with four GPUs

accuracy ultimately decreases. For DASO, the effect is more pronounced as completing batches without a global synchronization has a similar effect to increasing the size of the globally distributed batch. As the classic algorithm does not do any compression, it is reasonable to assume that the communication of the gradients in their full precision is beneficial to the accuracy of the network at all node counts.

Semantic segmentation—cityscapes

To further evaluate the performance of the DASO method, we conducted experiments on a cutting edge network. To this end, a hierarchical multi-scale attention network [14] was trained for semantic segmentation on the Cityscapes [40] dataset. This dataset is a collection of images of streets in 50 cities across the world, with 5,000 finely annotated images and 20,000 coarsely annotated images. The network has an HRNet-OCR backbone, a dedicated fully convolutional head, an attention head, and an auxiliary semantic head [14].

The quality of semantic segmentation networks is often evaluated based on the intersection over union (IOU) [41] score. IOU is defined as the intersection of the correctly predicted annotations with the ground truth annotations, divided by their union. The IOU ranges from 0.0 to 1.0, where higher values indicate more accurate predictions.

The network hyperparameters are shown in Table 3. The number of epochs, loss function, and optimizer settings were determined from the original source [14]. For the DASO experiments, the synchronized batch normalization operation is conducted within the node-local process group.

In its original publication, the network was trained using supplementary data, whereas the herein presented experiments are performed using only the Cityscapes dataset. To determine a baseline accuracy, the original network was trained with four GPUs on a single node using PyTorch's `DistributedDataParallel` package. This baseline measurement employed a polynomial decay learning rate scheduler, PyTorch's automatic mixed precision training and synchronized batch normalization layers. For more detail, see [14]. The baseline IOU of the original network was found to be 0.8258.

During the experiments, we found that for Horovod neither the automatic mixed precision nor the synchronized batch normalization functioned as intended when using the system scheduler software (SLURM [43]). Horovod requires usage of its custom scheduler, `horovodrun`, to enable full feature functionality. However, this software is not natively available on many computer clusters. Hence, automatic mixed precision was removed and the synchronized batch normalization layers were replaced with standard batch normalization layers.

In the interest of limiting CO_2 emissions, the wall clock time limit for each measurement was set to 15h. For DASO and Horovod, this was not a factor at any point in these measurements. However, the classic DPNN algorithm was extremely slow while attempting to train this model using this dataset. Therefore, the time required to train the model fully was extrapolated from the completed epochs. These results are shown in Fig. 7a. As the trainings were not able to be completed in a reasonable time, IOUs are not reported for the classic method

Training times for various node counts are shown in Fig. 7a. For up to 128 GPUs, DASO completed the training process in approximately 35% less time than Horovod, demonstrating the advantage of our approach to fully leverage the systems communication architecture together with asynchronous parameter updates. At higher GPU counts the time savings drop to 30%, because there are fewer batches per epoch, and hence skipping global synchronization operations provides less benefits. The classic algorithm is prohibitively slow for this experiment. It is between five and 31 times slower than Horovod and between eight and 45 times slower than DASO. The timing measurements of DASO and Horovod show the importance of using optimized data parallel training methods for training large models.

Quality measurements (IOU) are shown in Fig. 7b. Although there is a very clear difference between Horovod and DASO, neither matches the accuracy of the baseline network. This is due to the naïve learning rate scheduler used for training. With a tuned learning rate optimizer the 16, 32, and 64 node configuration should more accurately recreate the results of the baseline network. At 256 GPUs, training with Horovod did not yield any meaningful results. We hypothesize that this is caused by the lack of a functioning synchronized batch normalization operation in combination with a very large mini-batch.

Conclusions and future works

In this work, we have introduced the distributed asynchronous and selective optimization (DASO) method. DASO utilizes a hierarchical communication scheme to fully leverage the communications infrastructure inherent to node-based computer clusters, which often see multiple GPUs per node. By favoring node-local parameter updates, DASO is able to reduce the amount of global communication required for full data parallel network synchronization. Thereby, our approach alleviates the bottleneck of blocking synchronization used in traditional data parallel approaches. We show that, if independent and identically distributed (iid) batches can be reasonably assumed, the global synchronization ubiquitous to the training of DPNNs is not required after each forward-backward pass. Furthermore, stale network states can be used in conjunction with a reduced number of global synchronizations to accurately train classical and state-of-the-art networks.

In a parameter study, we demonstrated that the accuracy of a model depends strongly on how frequently the global parameters are synchronized and the number of devices used to train the network concurrently. This study also showed that stale gradients can be used to accurately train a network. However, the combined effects of stale gradients and selective global updates require preventative measures to ensure robust network architectures can be properly trained on large numbers of GPUs. Furthermore, this parameter study showed a very interesting relationship between the stale gradients and the local synchronizations. Namely, the accuracy steady decreased with increasing staleness until the number of batches between global synchronizations was equal to the number of batches to wait for the data, at which point it increased greatly. This effect should be studied in more depth, as it may provide greater insight into how neural networks are trained.

We evaluated DASO on two common DPNN use-cases: image classification on the ImageNet dataset with ResNet-50, and semantic segmentation on the Cityscapes dataset with a cutting edge multi-head attention network architecture. Our experiments show that DASO can reduce training time by up to 34% while maintaining similar prediction accuracy when compared to Horovod, the current standard for data parallel network training, and by up to 95% when compared with a classic synchronized SGD approach.

At large node counts, DASO and Horovod both suffer a decrease in network accuracy. This is a well-known problem which relates to an increase in the distributed batch size. The effect is more pronounced with DASO due to the reduced number of global synchronization steps. This allows for the identification of where network modifications must be employed to handle very large node counts. We also note that DASO and Horovod will both yield sub-optimal results on datasets for which the iid assumption no longer holds. For those cases, however, data parallel training will be ineffective regardless of the communications scheme. Overall, DASO achieves close-to-optimal target metrics significantly faster than Horovod. Therefore, DASO is optimal for rapid initial training of large networks, respectively datasets, where the training can be further fine-tuned using more traditional methods.

We have shown that DASO improves the scalability of DPNNs and demonstrates that using more GPUs does not have to be the only solution to speeding up training. While these results are very promising, there remain many things to explore in this direction. The parameter study showed that there are many effects which can benefit or detract from training a network at scale and that these effects need to be understood if we are to gain further insight on how to train networks on large numbers of devices.

DASO's advantage lies in the fact that it is a generic, non-tailored, and easy to implement approach that translates well to any large scale system, may it be a cloud, a node-based computer cluster or a high-performance computing system. DASO opens the door to redefining data parallel neural network training towards asynchronous, multifaceted optimization approaches.

Abbreviations
DPNN: Data parallel neural network; DASO: Distributed, asynchronous, and selective optimization; GPU: Graphics processing unit; SGD: Stochastic gradient descent; ASGD: Asynchronous stochastic gradient descent; iid: Independent and identically distributed; MPI: Message passing interface; NCCL: NVIDIA Collective Communication Library; Heat: Helmholtz analytics framework; ID: Identification; JUWELS: Jülich Wizard for European Leadership Science; AMD: Advanced Micro Devices; CPU: Central processing unit; HDR: High dynamic range; CUDA: Compute Unified Device Architecture; ResNet: Residual neural network; HRNet-OCR: High-Resolution Network-Object-Contextual Representations; IOU: Intersection over union; HPC: High performance computing; NN: Neural network; LR: Learning rate.

Supplementary Information

The online version contains supplementary material available at https://doi.org/10.1186/s40537-021-00556-1.

Additional file 1. Proof of convergence.

Acknowledgements
This work is supported by the Helmholtz Association Initiative and Networking Fund under project number ZT-I-0003, the Helmholtz AI platform grant and the HAICORE@KIT partition.

Authors' contributions
DC: Development and implementation of DASO, ran experiments, main text authorship. Assisted with implementation of the classic DPNN method mentioned in "Performance evaluation" section. Creation of figures CD: Main text authorship and extensive editing MG: Supervision of DC, assisted in algorithm design, extensive editing FL: Primary author of code for the classic DPNN implementation JK: Editing and considerable assistance in concept development for DASO MS: Supervisor of FL, assisted in algorithm design AS: Supervisor and group leader of DC, assisted in concept development and algorithmic design. All authors read and approved the final manuscript.

Funding
This work is supported by the Helmholtz Association Initiative and Networking Fund under project number ZT-I-0003, the Helmholtz AI platform Grant and the HAICORE@KIT partition. Open Access funding enabled and organized by Projekt DEAL.

Availability of data and materials
The datasets generated and/or analysed during the current study are available in the following repositories: ImageNet-2012:https://image-net.org/: https://www.cityscapes-dataset.com/.

Declaration

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

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

Author details
[1]Karlsruhe Institute of Technology, Hermann-von-Helmholtz-Platz 1, 76344 Eggenstein-Leopoldshafen, Germany. [2]German Aerospace Center, Linder Höhe, 51147 Cologne, Germany.

Received: 21 August 2021 Accepted: 22 December 2021
Published: 4 February 2022

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), pp. 770–778 (2016). https://doi.org/10.1109/CVPR.2016.90
2. Vaswani, A., Shazeer, N., Parmar, N., et al.: Attention Is All You Need. [accessed on 2021-08-06] (2017). arXiv:1706.03762.
3. Ben-Nun T, Hoefler T. Demystifying parallel and distributed deep learning: an in-depth concurrency analysis. ACM Comput Survey. 2019;52(4):1–43. https://doi.org/10.1145/3320060.
4. Lee S, Purushwalkam S, Cogswell M, et al. Why M heads are better than one: Training a diverse ensemble of deep networks. CoRR (2015). arXiv:1511.06314.

5. Krizhevsky A. One weird trick for parallelizing convolutional neural networks. CoRR **abs/1404.5997** (2014). arXiv:1404.5997.
6. Geng J, Li D, Wang S. Elasticpipe: An efficient and dynamic model-parallel solution to dnn training. In: Proceedings of the 10th Workshop on Scientific Cloud Computing. ScienceCloud '19, pp. 5–9. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3322795.3331463
7. Byungik Ahn J. Neuron machine: Parallel and pipelined digital neurocomputing architecture. In: 2012 IEEE International Conference on Computational Intelligence and Cybernetics (CyberneticsCom), pp. 143–147 (2012). https://doi.org/10.1109/CyberneticsCom.2012.6381635.
8. Zickenheiner S, Wendt M, Klauer B, Waldschmidt K. Pipelining and parallel training of neural networks on distributed-memory multiprocessors. In: Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94). 1994;4:2052–20574. https://doi.org/10.1109/ICNN.1994.374529.
9. Huang Y, Cheng Y, Bapna A, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In: Wallach, H., Larochelle, H., Beygelzimer, A., et al. (eds.) Advances in Neural Information Processing Systems, vol. 32 (2019). https://proceedings.neurips.cc/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf.
10. Liu Y, Yang J, Huang Y, Xu L, Li S, Qi M. Mapreduce based parallel neural networks in enabling large scale machine learning. Intell Neurosc. 2015. https://doi.org/10.1155/2015/297672.
11. Keuper J, Preundt FJ. Distributed training of deep neural networks: Theoretical and practical limits of parallel scalability. In: 2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC), pp. 19–26 (2016). https://doi.org/10.1109/MLHPC.2016.006.
12. Dryden N, Moon T, Jacobs SA, Van Essen B. Communication quantization for data-parallel training of deep neural networks. In: 2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC), pp. 1–8 (2016). https://doi.org/10.1109/MLHPC.2016.004.
13. Yamazaki M, Kasagi A, Tabuchi A, Honda T, et al. Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds. 2019. arXiv:1903.12650. Accessed 06 Aug 2021.
14. Tao A, Sapra K, Catanzaro B. Hierarchical Multi-Scale Attention for Semantic Segmentation. 2020. arXiv:2005.10821. Accessed 06 Aug 2021.
15. Sergeev A, Balso MD. Horovod: fast and easy distributed deep learning in TensorFlow. 2018. arXiv:1802.05799. Accessed 06 Aug 2021.
16. Bottou L, Curtis FE, Nocedal J. Optimization Methods for Large-Scale Machine Learning. ArXiv; 2018. Accessed 06 Aug 2021.
17. De Sa C, Feldman M, Ré C, Olukotun K. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In: Proceedings of the 44th Annual International Symposium on Computer Architecture. ISCA '17, pp. 561–574. Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3079856.3080248.
18. Lian X, Zhang W, Zhang C, Liu J. Asynchronous Decentralized Parallel Stochastic Gradient Descent. In: Proceedings of the 35th International Conference on Machine Learning (ICML); 2018, pp. 3043–3052.
19. Zhang S, Zhang C, You Z, et al. Asynchronous Stochastic Gradient Descent for DNN Training. In: 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, 2013, pp. 6660–6663. https://doi.org/10.1109/ICASSP.2013.6638950
20. Dutta S, Wang J, Joshi G. Slow and Stale Gradients Can Win the Race. 2020. arXiv:2003.10579. Accessed 06 Aug 2021.
21. Zhang W, Gupta S, Lian X, Liu J. Staleness-aware Async-SGD for Distributed Deep Learning. 2016. arXiv:1511.05950. Accessed 06 Aug 2021
22. Bogoychev N, Junczys-Dowmunt M, Heafield K, Aji AF. Accelerating Asynchronous Stochastic Gradient Descent for Neural Machine Translation. 2018. arXiv:1808.08859. . Accessed 06 Aug 2021.
23. Lin T, Stich SU, Jaggi M. Don't use large mini-batches, use local sgd. arXiv:1808.07217 (2020)
24. Paszke A, Gross S, Massa F, Lerer A, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: Advances in Neural Information Processing Systems 32, pp. 8024–8035 (2019). http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
25. Abadi M, Agarwal A, Barham P, et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Software available from tensorflow.org, 2015. http://tensorflow.org/. . Accessed 04 Aug 2021.
26. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Version 3.1, 2015. https://fs.hlrs.de/projects/par/mpi//mpi31/
27. Ueno Y, Yokota R. Exhaustive Study of Hierarchical AllReduce Patterns for Large Messages Between GPUs. In: 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pp. 430–439 (2019). https://doi.org/10.1109/CCGRID.2019.00057.
28. Mikami H, Suganuma H, U Chupala P, et al. Massively Distributed SGD: ImageNet/ResNet-50 Training in a Flash. 2018. arXiv:1811.05233. Accessed 06 Aug 2021.
29. Clauset A. A Brief Primer on Probability Distributions. 2011. http://tuvalu.santafe.edu/aaronc/courses/7000/csci7000-001_2011_L0.pdf. Accessed 06 Aug 2021.
30. Li A, Song SL, Chen J, et al. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. IEEE Trans Parallel Distrib Syst. 2020;31(1):94–110. https://doi.org/10.1109/tpds.2019.2928289.
31. Alistarh D, Grubic D, Li J, Tomioka R, Vojnovic M. QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding; 2017. arXiv:1610.02132. Accessed 06 Aug 2021.
32. Götz M, Debus C, Coquelin D, Krajsek K, Comito C, Knechtges P, Hagemeier B, Tarnawa M, Hanselmann S, Siggel M, Basermann A, Streit A. HeAT - a Distributed and GPU-accelerated Tensor Framework for Data Analytics. In: 2020 IEEE International Conference on Big Data (Big Data), pp. 276–287 (2020). https://doi.org/10.1109/BigData50022.2020.9378050
33. Li S, Zhao Y, Varma R, et al. PyTorch Distributed: Experiences on Accelerating Data Parallel Training; 2020. arXiv:2006.15704. Accessed 06 Aug 2021.
34. Krause D. JUWELS: Modular Tier-0/1 Supercomputer at the Jülich Supercomputing Centre. J Large-scale Res Facil. 2019;5:135. https://doi.org/10.17815/jlsrf-5-171.

35. NVIDIA: NVIDIA A100 TENSOR CORE GPU. (2021). NVIDIA. https://www.nvidia.com/en-us/data-center/a100/.
36. Deng J, Dong W, Socher R, Li LJ, et al. ImageNet: A Large-scale Hierarchical Image Database. In: 2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 248–255 (2009). https://doi.org/10.1109/CVPR.2009.5206848.
37. NVIDIA Corporation: NVIDIA Data Loading Library (DALI); 2021. https://developer.nvidia.com/DALI. Accessed 05 Aug 2021.
38. Goyal P, Dollár P, Girshick R, Noordhuis P, et al. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour; 2018. arXiv:1706.02677. Accessed 06 Aug 2021.
39. Wu Y, Kirillov A, Massa F, et al. Detectron2. https://github.com/facebookresearch/detectron2; 2019.
40. Cordts M, Omran M, Ramos S, et al. The Cityscapes Dataset for Semantic Urban Scene Understanding. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 3213–3223 (2016). https://doi.org/10.1109/CVPR.2016.350.
41. Rezatofighi H, Tsoi N, Gwak J, et al. Generalized Intersection over Union: A Metric and A Loss for Bounding Box Regression; 2019. arXiv:1902.09630. Accessed 06 Aug 2021.
42. Zhao S, Wang Y, Yang Z, Cai D. Region Mutual Information Loss for Semantic Segmentation; 2019. arXiv:1910.12037. Accessed 06 Aug 2021.
43. Yoo AB, Jette MA, Grondona M. SLURM: Simple Linux Utility for Resource Management. In: Workshop on Job Scheduling Strategies for Parallel Processing, pp. 44–60 (2003). https://doi.org/10.1007/10968987_3.

Publisher's Note