The Journal of Supercomputing
https://doi.org/10.1007/s11227-020-03200-6

Training deep neural networks: a static load balancing approach

Sergio Moreno-Álvarez[1] · Juan M. Haut[2] · Mercedes E. Paoletti[2] · Juan A. Rico-Gallego[1] · Juan C. Díaz-Martín[2] · Javier Plaza[2]

Abstract
Deep neural networks are currently trained under data-parallel setups on high-performance computing (HPC) platforms, so that a replica of the full model is charged to each computational resource using non-overlapped subsets known as batches. Replicas combine the computed gradients to update their local copies at the end of each batch. However, differences in performance of resources assigned to replicas in current heterogeneous platforms induce waiting times when synchronously combining gradients, leading to an overall performance degradation. Albeit asynchronous communication of gradients has been proposed as an alternative, it suffers from the so-called staleness problem. This is due to the fact that the training in each replica is computed using a stale version of the parameters, which negatively impacts the accuracy of the resulting model. In this work, we study the application of well-known HPC static load balancing techniques to the distributed training of deep models. Our approach is assigning a different batch size to each replica, proportional to its relative computing capacity, hence minimizing the staleness problem. Our experimental results (obtained in the context of a remotely sensed hyperspectral image processing application) show that, while the classification accuracy is kept constant, the training time substantially decreases with respect to unbalanced training. This is illustrated using heterogeneous computing platforms, made up of CPUs and GPUs with different performance.

Keywords Deep learning · High-performance computing · Distributed training · Heterogeneous platforms

✉ Sergio Moreno-Álvarez
smoreno@unex.es

Extended author information available on the last page of the article

Published online: 02 March 2020

1 Introduction

Deep learning (DL) algorithms based on neural network architectures [19] have reached great accuracy in areas such as image classification [17, 20] and speech recognition [5] among others. When compared with other machine learning (ML) and pattern recognition methods, deep neural networks (DNNs) work as universal approximators of parameterized maps (models) composed of stacks of layers [12], where each one is composed by several nodes (neurons) connected to the nodes of the precedent and subsequent layers through synaptic weights and saturation control biases [22]. Overall, DNN models fit neuron weights and biases through an iterative optimization process based on training with examples. Improvements with respect to traditional techniques are supported by the large amount of data available to train these models, as well as by advances in high-performance computing (HPC) platforms [9].

DNN learning strategies can be roughly classified into supervised and unsupervised learning [14], depending on whether they use labeled data or not. This work focuses on supervised image classification with DNNs, whose input dataset, $\mathcal{X}$, is composed of n_{examples} images of $\mathbf{x}_n \in \mathbb{R}^{h\times w\times c}$ $(n = 1, \dots, n_{\text{examples}})$, where $h \times w$ denotes the spatial dimensions of the images (i.e., height and width) and c the spectral depth (i.e., the number of channels). $\mathcal{X}$ is divided into two main subsets. The first one is the so-called training set, on which the classifier adjusts its weights and biases. The second set is the test set, on which the classifier makes the inference. With this in mind, it is easy to describe the DNN for image classification as a mapping model $\mathcal{M}(\cdot, \theta)$ with parameter θ that performs $\mathcal{M} : \mathcal{X} \rightarrow \mathcal{Y}$, associating each image of the original dataset ($\mathcal{X}$) with a corresponding label ($\mathcal{Y}$) by adjusting the parameters of the model θ. On this wise, the purpose of supervised learning is to find optimal $\theta^{\star}$ values in order to minimize the distance between the outputs of the model and the labeled values. Such distance is determined by the so-called *loss function* $L(\theta)$ (e.g., mean square error) during the training stage. The *Stochastic Gradient Descent* (SGD) method is commonly used for this purpose. In each iteration k, SGD updates θ through gradient g^k of $L(\theta)$ as $\theta^{k+1} = \theta^k - \alpha\, g^k$, where α is the so-called *learning rate*, which controls the advance in the weight domain. g^k is computed as $g^k = \frac{1}{|\chi|} \sum_{n\in\chi} \nabla l(n|\theta^k)$, being $l(n|\theta^k)$ the loss of the example n, computed on θ^k .

The updating process demands high computational capacity. To accelerate it, dedicated HPC clusters and non-dedicated cloud platforms are commonly used for large-scale deep networks training on large datasets. The training process is usually distributed on the platform resources based on two main schemes of parallelization, known as *model parallelism* and *data parallelism.*

Model parallelism is used when the model is too large to fit in the memory of an isolated computational resource, and hence, it is split and deployed among the available resources. Every process trains its own portion of the model using the same batch of examples. Depending on the deployment of the model, learners communicate intermediate results using different strategies [13]. As training is an inherently sequential process, this scheme could result in the underexploitation of the computational resources, as its performance is limited by the communication between the processes and by the number of nodes involved in the training process.

Conversely, data parallelism consists of running *replicas* of the training model on the available computational resources. Every replica holds a local copy of the entire model, which is trained on disjoint data subsets of χ called *batches* [18]. After every batch in the training step is completed, replicas compute the gradients of their respective *loss* function. Next, they coordinate to combine their local computed gradients before updating θ, commonly by tree reduction collective communications [23], or pushing gradients to centralized *parameter servers* [7]. Needless to say, these synchronization points cause straggler processes to have a high impact on the overall performance. Furthermore, performance degradation grows with the heterogeneity of the platform and the number of replicas used to train the model. Model and data types of parallelism can be combined in order to mitigate their limitations in some particular scenarios. This is known as *hybrid parallelism*.

Two general mechanisms contribute to solve the aforementioned performance issue. First, using an asynchronous SGD optimization procedure to relax the consistency of parameter values by allowing processes to asynchronously combine gradients and update parameters. This scheme decouples communication and computation, which highly benefits the training performance. However, the order in which parameters are updated is not deterministic, and hence gradient computations in a replica are done on a stale version of parameter values. This *distance* between a local parameter used to compute gradients in a replica and its global current value is known as *staleness*. Empirical studies [7] show that a low degree of staleness does not penalize the learning accuracy of the model. Meanwhile, other works [10] propose mechanisms for reducing staleness impact on the accuracy of training models.

A second approach to mitigate staleness consists of using load balancing techniques to assign to each replica an amount of work which is in accordance with its computational capacity. A dynamic load balancing mechanism is proposed in [3], in which, in each epoch, every replica is assigned with a batch size proportional to its speed. A key point is to determine the speed of each replica in the system, and this is achieved by using an additional recurrent neural network that calculates the size of the batch in the next epoch as a function of the current speed and processing time. A non-dedicated cloud platform is assumed in this context, with shared assigned computing resources and, hence, variable speed and memory parameters. Although adaptive (and partially able to deal with stragglers due to temporal speed variations), this mechanism requires a large number of epochs to be effective and steals computational resources from the main training process.

In this paper, we introduce a new mechanism for improving the distributed performance of the training process of deep models, while keeping their accuracy. Our methodology follows a two-step approach. First, prior to the training, we use the FuPerMod tool [6] to determine the computational capabilities of the computational resources assigned to each replica in the platform. We assume dedicated heterogeneous clusters made up of both CPUs and GPUs. Secondly, in the training step, we assign a batch size to each replica that is proportional to its relative speed. The goal is to balance the workload and, as a consequence, to minimize the communication of the waiting times of the replicas at the time of communicating the computed gradients. We use the synchronous SGD with reduction tree communication to combine gradients, which ensures deterministic order in combining the per-replica computed gradients, and hence it does not affect the overall accuracy.

The static load balancing technique has been widely used in the HPC field [1, 21], because it does not require any additional computational resources during the execution time of an application. Notwithstanding this fact, especially in non-dedicated platforms where the workload variation is higher, it can be combined with other mechanisms, such as those proposed in [3] to perform fine-grained and dynamic adaptations during the training stage. Furthermore, this can be combined with asynchronous SGD techniques, such in [4], to ensure minimal staleness. Hence, the primary contributions of this work are:

- A new methodology to improve the performance of data-parallel distributed training of deep models, while preserving the accuracy of the trained model when replicas run on dedicated heterogeneous platforms.
- The application of common HPC-based static workload balancing techniques for training deep models, in order to optimize resource exploitation and execution times.
- An evaluation of the impact of the heterogeneity of dedicated computing platforms on the deep network distributed training process.

The rest of this paper is organized as follows. First, we discuss related works in Sect. 2. Section 3 describes our implementation, including the distribution of processes on the heterogeneous platform and the training model procedure. Section 4 details how we evaluated our system and presents the obtained results. Finally, Sect. 5 presents our conclusions and future work.

2 Related work

The increase in dataset sizes and the number of parameters to learn in deep models have leveraged the usage of HPC platforms to accelerate training, including dedicated clusters and non-dedicated cloud environments. The work in [2] provides an excellent survey of current distributed techniques to parallelize and distribute the training. The main data and model parallelization schemes are described in work [16] that proposes a distribution training of convolutional networks [19] using data parallelism in convolutional layers and model parallelism in fully connected layers, as well as different synchronization methods for parameter updating between workers. In this paper, we focus on the data parallelism scheme applied to convolutional networks.

Paper [7] proposes the *Downpour* asynchronous SGD algorithm implemented in the *DistBelief* framework. This framework enables large-scale model and data parallelism for training purposes. The Downpour algorithm launches multiple replicas of the model using data parallelism, and it uses asynchronous SGD gradient communication based on a centralized parameter server. Authors empirically found that reaching a certain level of staleness tolerance does not significantly impact the model accuracy. Nevertheless, work [10] proposes a mechanism for reducing the staleness of the asynchronous SGD by modulating the learning rate using the current average gradient staleness values. They provide a discussion on the interplay of hyperparameters and distribution design choices, using the implemented *Rudra* framework. The staleness problem is also addressed in [4], using a different approach. This work performs data-parallel training of models by using *p backup workers* in addition to P replicas and data-parallel synchronous optimization. In order to update the model parameters, it considers P faster gradient computations and drops the rest. This approach reduces staleness and performance degradation caused by straggler replicas, at the expense of using additional resources. In this paper, we follow a synchronous SGD approximation method that avoids the staleness problem, although it is more sensitive to waiting times at synchronization.

Focusing on heterogeneous platforms, work [15] studies the performance degradation of SGD optimization in heterogeneous platforms (with respect to homogeneous distributed training schemes). They focused on a *Stale Synchronous Parallel* synchronization scheme, in which the grade of staleness is limited by the updating protocol and the parameter server. The authors propose both constant and dynamic learning rate schedules for updating the parameters. In this way, the unstable convergence caused by stragglers is mitigated, improving statistical and hardware performance. A key difference with respect to our work is that we use collective communication as synchronization mechanism, avoiding the necessity of additional parameter server processes.

The thorough work in [3] proposes to adapt batch sizes in each replica to their relative speeds. As a consequence, straggler replicas waiting times are minimized. Authors use a *Bulk Synchronous Parallel* scheme (which avoids staleness) on heterogeneous non-dedicated cloud platforms, with simulated injected stragglers in their experiments. The measurement of the respective speed of the replicas (needed to compute their assigned batch sizes) is achieved using a *Recurrent Neural Network*, trained along epochs with per-replica CPU performance and memory usage values in each iteration. It is worth noting that our work follows a similar approach to those adapting batch sizes in replicas to their computational performance. However, a novelty of our work is to introduce an offline and static load balance mechanism which does not interfere with the training of the model. Work [3] offers additional insights that we plan to include in future works, such as conducting weighted gradient aggregation to avoid per-sample biases in replicas.

3 Detailing the adopted approach

This section details our proposal for training of deep models[1] following a data-parallel scheme. We assume a dedicated heterogeneous platform made up of a set of computational *nodes*, with different speeds. Nevertheless, our approach can also be applied to non-dedicated cloud environments as an initial workload distribution,

[1] The source code is available at https://github.com/mhaut/static_load_deeplearning.

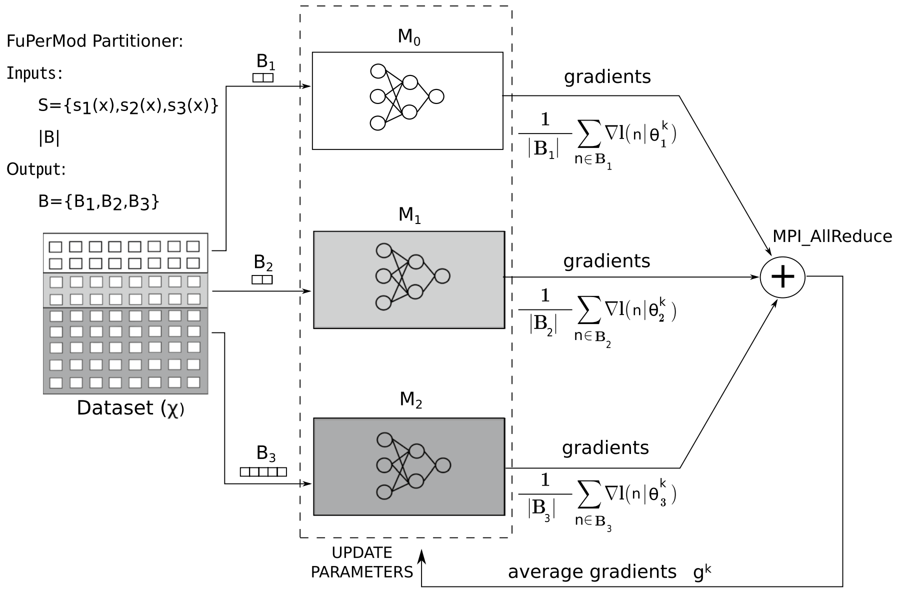
FuPerMod Partitioner:
Inputs:
S={s1(x),s2(x),s3(x)}
|B|
Output:
B={B1,B2,B3}
Dataset (χ)
B1
B2
B3
M0
M1
M2
gradients
$\frac{1}{|B_1|}\sum_{n\in B_1}\nabla l(n|\theta_1^k)$
$\frac{1}{|B_2|}\sum_{n\in B_2}\nabla l(n|\theta_2^k)$
$\frac{1}{|B_3|}\sum_{n\in B_3}\nabla l(n|\theta_3^k)$
MPI_AllReduce
UPDATE PARAMETERS
average gradients g^k

Fig. 1 General distributed learning framework of a model using static load balance mechanisms. The figure represents an iteration k of an epoch in the training process, where $P = 3$ replicas are assigned with an uneven batch size B_p. Replicas perform the parameter update after communicating gradients (g^k) using the `MPI_Allreduce` collective operation

or assuming the possible variations in performance. Replicas running the training process are deployed on the computing nodes of the platform, commonly multicore CPUs and GPUs with different numbers of cores and processing speeds on current heterogeneous platforms.

A key issue is to accurately determine the speed of each node. In this regard, FuPerMod is a tool that is commonly used in heterogeneous HPC platforms. It determines empirically the computational capabilities of a node running a given application. To achieve that, the tool executes a benchmark (provided by the user) in each compute node. This benchmark should be representative of the application in order to obtain meaningful execution profiles. In this work, we use a convolutional neural network (on a range of batch sizes) as the benchmark to measure the speed in each replica. As output, FuPerMod returns the speed of the P computational nodes in the heterogeneous platform, as a set of P functions $S = \{s_1(x), s_2(x), \dots, s_P(x)\}$. A function $s_p(x)$ varies along a given range of batch sizes x to represent the performance profile of a computational node, which depends on its available resources (including cache and memory sizes). FuPerMod speed is provided as the inverse of the time invested in executing the benchmark for a given batch size $|B|$.

It is important to note that the set of speed functions S characterizing the platform is statically determined in a previous initialization step. Observe Fig. 1. The speed functions, together with the specific batch size $|B|$, are used as inputs to the FuPerMod *partitioner* utility, which computes the number of examples $|B_p|$ assigned to each replica p, with $\sum_{p=1}^{P} |B_p| = |B|$. As FuPerMod *partitioner* works with sizes in bytes, we convert the input batch size to bytes using $h \times w \times c \times r \times |B|$, with r the number of bytes to represent each pixel.

To train the model, SGD works as a synchronous iterative process over multiple passes of the dataset, called *epochs*. In each epoch, the dataset χ is split up in P subsets and assigned to the P corresponding replicas. The size of the subset in a replica p is computed as $|\chi|/|B| \times |B_p|$. Furthermore, the distribution of the dataset to the replicas is made in such a way that batches assigned to replicas do not overlap in each epoch and, additionally, every replica trains its own copy of the model on the full set of examples along epochs. Then, replicas use their assigned subsets in batches to train their copies of the model. A description of the process is shown in Fig. 1. Iteration k of the SGD training process performs the following general steps:

1. Every replica p manages a batch of $|B_p|$ examples, with $|B| = \sum_{p=1}^{P} |B_p|$. The size of the batch B_p is proportional to the relative speed of the replica p, running on computational resources with speed $s_p(|B_p|)$.
2. Replicas compute their gradients based on a *Loss* function $l(n|\theta_p^k)$ that returns the error of the sample $n \in B_p$, computed in the iteration k using parameters θ_p^k with respect to the actual value. The gradient computation in a replica p is $g_p^k = \frac{1}{|B_p|} \sum_{n \in B_p} \nabla l(n|\theta_p^k)$.
3. After computing the gradient vectors g_p^k, each replica delivers a collective all-reduce communication operation in order to combine gradients as: $g^k = \frac{1}{P} \sum_{p=1}^{P} g_p^k$
4. Each replica updates the parameters using the received gradients and α.

We use PyTorch to train a model based on the previous steps. It implements MPI blocking collectives [8], which impose synchronization, and hence, straggler processes degrade performance. The proposed methodology is also compatible with the use of a *parameter server* that holds an updated global copy of the parameters The main drawback of this approach is its centralized nature that is mitigated by distributing parameters on several server processes [7] and the resources consumed by such additional server processes.

Finally, balancing the workload according to the replicas that determined computational capacity ensures that all replicas finish iteration k at the same time, avoiding idle times at the communication step and hence improving the overall performance. Of course, an error in the workload balancing may always exist, depending on errors in the measurement of the speed of the replicas using benchmarking, and on the granularity of the example size, which we consider negligible. A limitation of this methodology is that a static workload assignment to replicas does not adapt to temporal changes in the system loads derived from the shared usage of resources in non-dedicated platforms. We indeed assume a homogeneous network, and hence, we do not account for the influence of possible imbalances in the gradients communication performance in the model training.

At this point, it is important to note that [3] presents a weighted aggregation in the gradient computation for assigning every example with the same worth, as $g^k = \frac{1}{|B|} \sum_{p=1}^{P} g_p^k$. This improvement does not affect the performance, although it certainly has an impact in the convergence of the model.

4 Experimentation and analysis

This section evaluates the proposed approach. We detail first the hardware and software elements, and then we discuss the results of the training tasks in terms of performance and accuracy. To conclude, we experimentally test the proposed load balanced implementation in ResNet [11], a relevant and standard architecture in the field.

A small test platform called *Metropolis* is used to obtain initial insights and results of the behavior of the implementation and also a dedicated heterogeneous HPC cluster called *Ceta-Ciemat* to obtain real and more complete results of our implementation in a real environment. *Metropolis* is composed of two nodes, *Pluton* and *Caronte*, that use three different GPUs: an NVIDIA RTX 2080Ti and an NVIDIA RTX 2060 (connected to *Pluton*) and an NVIDIA RTX 1050Ti (connected to *Caronte*), with 11 GB, 6 GB and 4 GB of memory, respectively. The CPUs are an Intel Xeon E5620 (Nehalem) 8-core processors running at 2.40 GHz with 12 GB of RAM. Nodes are connected by an Ethernet GigaBit network. Five replicas are deployed in that set of computational nodes. A CPU core is reserved for each replica running on a GPU, in order to be used for memory transfers and communications. On the other hand, the *Ceta-Ciemat* cluster is composed of eight multicore CPU nodes connected by an InfiniBand QDR Network. Each CPU node holds one or two Kepler K80 GPUs with 24 GB of RAM. The CPUs are a 24-core Intel Haswell running at 2.50 GHz with 64 GB of RAM. Although all GPUs are similar, we experimentally found significant differences in speed between nodes using one GPU with respect to nodes using two GPUs. The reason is that GPUs share the PCI bus, which impacts the performance of data transfers between CPUs and GPUs. Such subtle performance penalties can be found in other platforms.

We trained our models using two standard datasets, *MNIST* and *CIFAR-10*. MNIST is composed of black and white handwritten digits images, representing digits from 0 to 9, with a training set of 60.000 examples and a test set of 10.000 examples. These digits have been normalized in size and centered to a fixed image size of $28 \times 28 \times 1$ of 32-bit floats. CIFAR-10 contains 60.000 color images of size $32 \times 32 \times 3$ of 10 non-biased classes. This dataset is used to verify that the proposed implementation provides successful results.

Two models are used in our experiments. Both have a feature extractor composed of several stages of convolutional and pooling layers and a classifier with fully connected (FC) layers. Details are shown in Tables 1 and 2. The two-dimensional convolutional layers have been implemented to extract deep features, employing the rectified linear unit (ReLU) activation function. The feature maps obtained in the convolutional part are reshaped into an unrolled vector representation to feed the classifier.

As a conduit example, we evaluate the performance of training the model described in Table 1 with the MNIST dataset in the *Metropolis* platform. Performance is measured using wall clock time per epoch. The final results are obtained by taking the maximum training time per epoch of the replicas involved

Table 1 Layers of the convolutional neural network implemented for classification of the MNIST image dataset

Model for MINST (Number of parameters: 21,840)				
Layer ID	Kernel/neurons	Activation funct.	Pooling	Dropout
Conv1	$5 \times 5 \times 10$	ReLU	2×2	No
Conv2	$5 \times 5 \times 20$	ReLU	2×2	No
FC1	50	ReLU	–	Yes
FC2	n_{classes}	Softmax	–	No

Padding is added to convolutional layers for not shrinking the image

Table 2 Layers of the convolutional neural network for the classification of the CIFAR-10 image dataset

Model for CIFAR-10 (Number of parameters: 176,034)				
Layer ID	Kernel/neurons	Activation funct.	Pooling	Dropout
Conv1	$7 \times 7 \times 10$	ReLU	2×2	No
Conv2	$7 \times 7 \times 20$	ReLU	2×2	No
FC1	120	ReLU	–	No
FC2	84	ReLU	–	No
FC3	n_{classes}	Softmax	–	No

Padding is added to convolutional layers to avoid shrinking the images

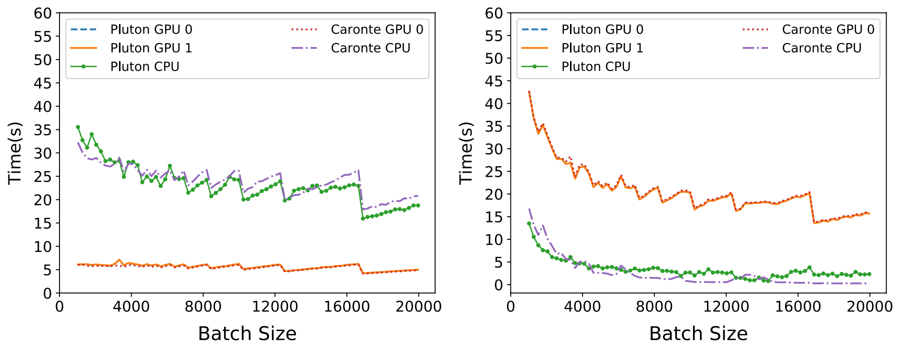

Fig. 2 Computation (left) and communication (right) times of training the model described in Table 1 on the MNIST dataset in *Metropolis* platform for a range of batch sizes, evenly distributed between $P = 5$ replicas. The processing time is provided in seconds per epoch

in the training process, along five different executions. To obtain our performance data, we run $e = 10$ epochs. As a baseline test, we use a homogeneous (unbalanced) distribution of the batch size $|B|$ between replicas in the platform, with $|B_p| = |B|/P, \forall p$. Figure 2 shows the performance data in separate computation and communication plots, for a range of batch sizes $|B|$. The communication time is measured from the invocation of the blocking communication operation to the reception of the gradients, and it includes waiting times. As expected, replicas

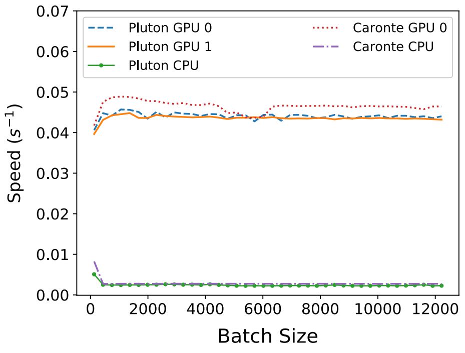

Fig. 3 FuPerMod characterization of the computational capabilities of $P = 5$ replicas running on the *Metropolis* nodes. Speeds are shown for a range of batch sizes $|B|$

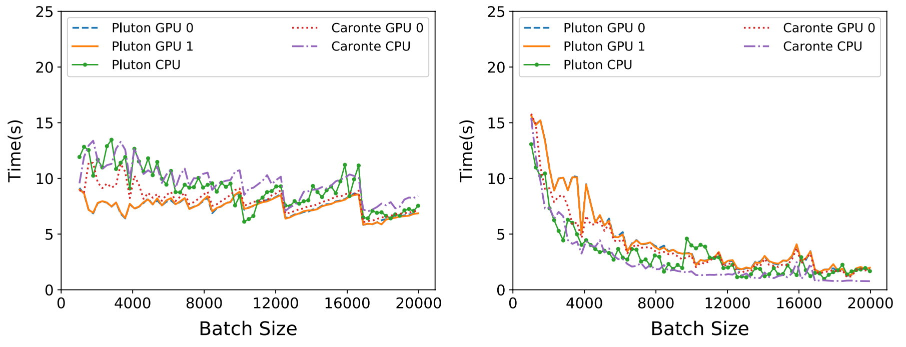

Fig. 4 Computation (left) and communication (right) times of training the model described in Table 1 on the MNIST dataset in *Metropolis* platform for a range of batch sizes. Batch sizes are unevenly distributed between $P = 5$ replicas, according to their speeds determined by FuPerMod tool. Time is provided in seconds per epoch

running on CPUs spend more time in computing their batches of examples than those on GPUs, because of their lower computing throughput. As a consequence, GPU replicas wait at communication points, degrading overall performance of the training process.

Figure 3 illustrates the speeds of the replicas running on the *Metropolis* computational nodes, obtained by benchmarking using FuPerMod. As expected, high differences between GPUs and CPUs are observed, with slight differences between processes running in similar resources (either CPUs or GPUs). Figure 4 shows the results of the training execution for a range of batch sizes $|B|$, under the proposed static balanced distribution. FuPerMod *partitioner* unevenly distributed every batch B between replicas according to their speeds. Due to the fact that every replica needs to complete a similar amount of computing work, communication waiting times are dramatically reduced with respect to Fig. 2. This reduction also arises from the reduction in the differences of the time lines in the plots. As a consequence, the overall performance is significantly improved, as shown in Fig. 5.

We now focus on the *Ceta-Ciemat* system to validate performance results obtained in the previous proof-of-concept *Metropolis* platform. Figure 6 (top)

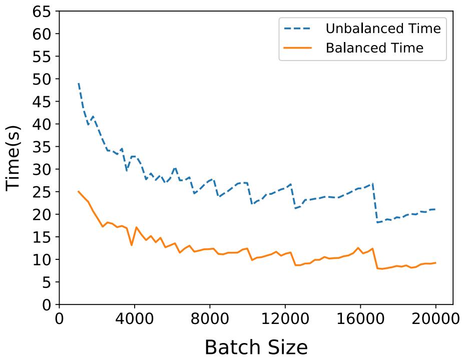

Fig. 5 Performance of training the model described in Table 1 on the MNIST dataset for unbalanced and balanced distributions between $P = 5$ replicas in *Metropolis*

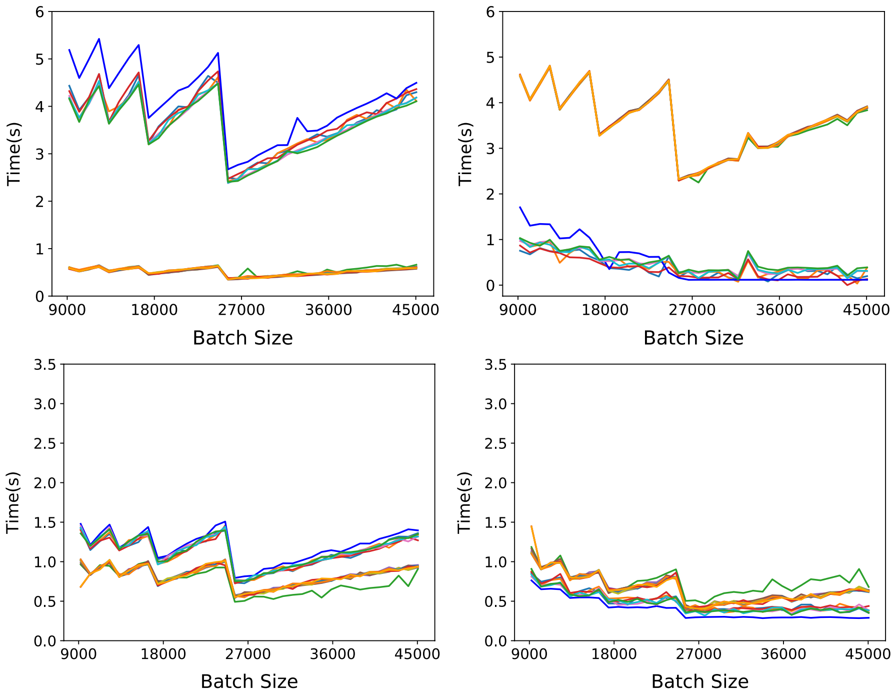

Fig. 6 Performance times in seconds per epoch of the computation (left) and communication (right) of training the model in Table 2, using CIFAR-10 dataset in the *Ceta-Ciemat* cluster, for increasing batch sizes. Top figures represent evenly (unbalanced) distribution and bottom figures represent unevenly (balanced) distribution between $P = 16$ replicas

shows the performance results obtained after training the model detailed in Table 2 with the CIFAR-10 dataset. We use a homogeneous distribution of the batch size, hence with $|B_p| = |B|/P, \forall p$, between $P = 16$ replicas deployed on available computational nodes of the platform. As in the previous platform, we have two groups of replicas depending on whether they run on CPUs or GPUs. Differences in performance between groups are high; however, slight differences

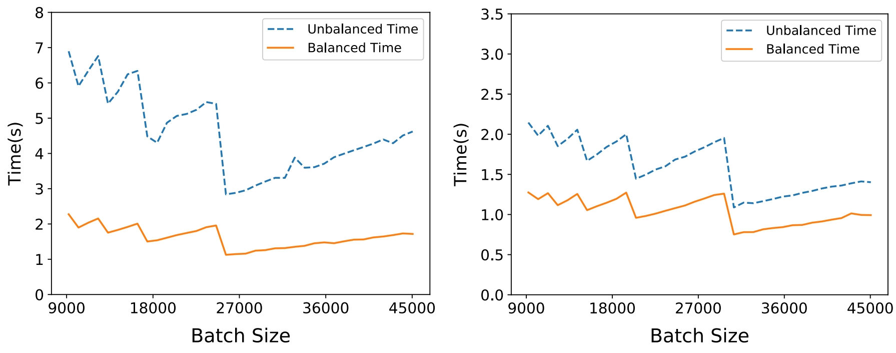

Fig. 7 Performance times in seconds per epoch for models trained with CIFAR-10 (left) and MNIST (right) datasets on the *Ceta-Ciemat* platform ($P = 16$ replicas) for unbalanced and balanced distributions of batch sizes between $P = 16$ replicas

in performance between replicas running on similar resources also impact the performance of the entire training process. Note that differences in computation (and hence, inversely, in communication, due to waiting times) grow with the batch sizes. The corresponding results obtained with a balanced distribution of the batch $|B|$ between replicas, according to their speeds, are shown at the bottom of Fig. 6. Again, plot lines differences shrink, meaning that replicas invest similar times in performing their assigned computing workloads, hence reducing waiting times at communication points. Figure 7 displays the total performance times for the two models in Tables 1 and 2, trained, respectively, on CIFAR-10 and MNIST on the *Ceta-Ciemat* platform with $P = 16$ replicas. Differences between unbalanced and balanced distributions remain constant along batch sizes for both MNIST and CIFAR-10 datasets. As a summary, the average speedups along the range of batch sizes for statically balanced workload distributions—with respect to unbalanced ones—are 1.52% and 2.78%, respectively.

Additionally, we evaluate the *accuracy*, defined as the percentage of correct classification of examples, given non-biased datasets as MNIST and CIFAR-10. In order to show the behavior of the models in both unbalanced (even) and balanced (uneven) workload distributions, the batch size is set to $|B| = 2048$ examples. Figure 8 shows both per-epoch and temporal evolution of the accuracy of the model trained on MNIST. We include the temporal evolution of accuracy to illustrate differences in convergence times of the methods with respect to training time. We can observe that both distribution approaches require around $e = 50$ epochs to converge to a similar accuracy. Figure 9 shows the corresponding results for CIFAR-10, with the difference that the model needs $e = 300$ epochs to converge on this (larger) dataset. We actually executed a higher number of epochs than those shown in both models plots; however, we experimentally observe that those values of e were enough for the models to converge. Table 3 summarizes the accuracy of the models including their standard deviation along five executions. The table shows the accuracy reached for MNIST and CIFAR-10 datasets in the *Ceta-Ciemat* platform for $e = 50$ and $e = 300$ epochs, respectively. Accuracy figures fall in the same range if we consider

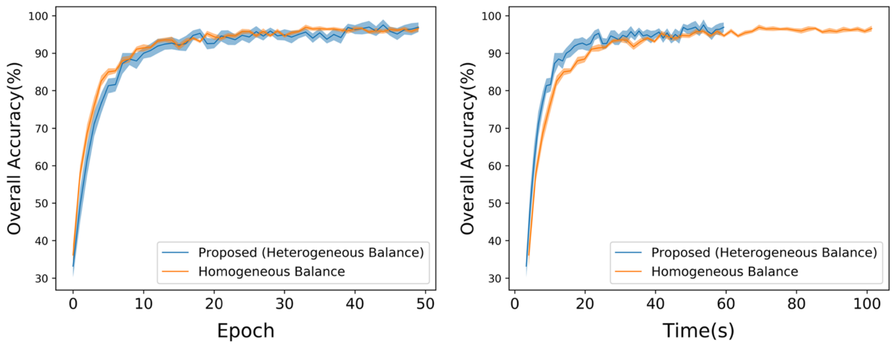

Fig. 8 Accuracy of the model described in Table 1, trained on MNIST dataset for unbalanced and balanced distributions of batch sizes |*B*| in the *Ceta-Ciemat* platform. Both per-epoch and temporal evolution of the accuracy are shown. The shaded areas represent the standard deviation along a set of five repetitions

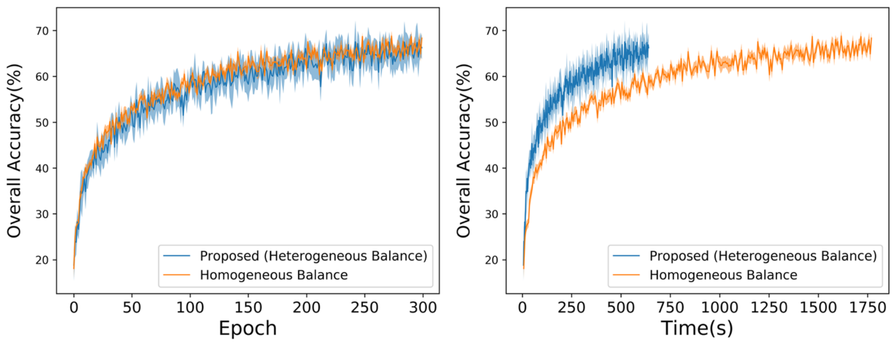

Fig. 9 Accuracy of the model described in Table 2 trained on CIFAR-10 dataset for unbalanced and balanced distribution of batch sizes |*B*| in the *Ceta-Ciemat* platform. Both per-epoch and temporal evolution of the accuracy are shown. The shaded areas represent the standard deviation along a set of five repetitions

Table 3 *Maximum* accuracy and *Last* accuracy (obtained in the last epoch) of models described in Table 1 and Table 2, trained on MNIST and CIFAR-10 datasets, respectively

Dataset	Maximum accuracy		Last accuracy		#Epochs
	Unbalanced	Balanced	Unbalanced	Balanced	
MNIST	**97.54** ± 1.47	96.92 ± 0.69	**96.92** ± 1.26	96.58 ± 0.75	$e = 50$
CIFAR-10	67.54 ± 4.64	**68.80** ± 1.11	66.22 ± 2.45	**68.34** ± 1.10	$e = 300$

The best metric values are in bold

the standard deviation; hence, there are no significant differences. Nevertheless, the aggregation of weighted gradients in the trained process (proposed in work [3] and described in Sect. 3) should make accuracy values even closer, without affecting

Table 4 Training times for several ResNet architectures

Network	Parameters (M)	Unbalanced	Balanced	Speedup
ResNet20	0.27	65.31	**25.67**	2.54
ResNet32	0.46	103.84	**41.76**	2.49
ResNet44	0.66	145.10	**58.25**	2.49
ResNet56	0.85	185.64	**74.24**	2.50
ResNet110	1.72	853.50	**104.07**	8.20

The best metric values are in bold

Parameters column shows the number of trainable parameters (in millions). Times for unbalance and balance workload distributions are in seconds per epoch. Last column shows obtained speedups

performance. Finally, as the previous results show that the accuracy remains constant with load balancing, performance tests have been carried out on the main ResNet architectures under CIFAR-10. Table 4 shows that the speedup obtained remains constant when the number of parameters is not triggered, while with a higher number of parameters it achieves a higher speedup.

5 Conclusions and future work

This paper describes a new approach to distribute the training of deep networks on dedicated heterogeneous platforms. The proposed methodology departs from a precomputed characterization of the speed of the computational resources of the platform. It provides replicas of the model (running in such computational resources) with a batch size that is proportional to their computing capabilities, in such a way that waiting times at communication points (performed at each training iteration to combine gradients between processes) are eliminated. As a consequence, the overall execution time needed for training the deep model is reduced (while the accuracy is not significantly affected), as demonstrated for two different platforms and datasets.

One of the main contributions of this work is the exploitation of well-known HPC optimization techniques to balance the distributed training of deep learning models. Previous works propose dynamic load balancing techniques that impact the performance, albeit requiring a high number of iterations to be really effective. In turn, our methodology can dynamically adapt batch sizes to performance variations during the training process, as a result from the non-dedicated usage of resources (which is characteristic of cloud environments).

Our future work will focus on the study of the scalability of the proposed implementation regarding the number of replicas, using both larger datasets and deeper networks.

Acknowledgements This work was jointly supported by the following projects and institutions: (1) The European Regional Development Fund 'A way to achieve Europe' (ERDF) and the Extremadura Local Government (Ref. IB16118). (2) The Ministry of Education, November 19, 2015, of the Secretary of State for Education, Vocational Training and Universities, under grant FPU15/02090. (3) The computing facilities of Extremadura Research Center for Advanced Technologies (CETA-CIEMAT), funded by the European Regional Development Fund (ERDF).

References

1. Beaumont O, Boudet V, Rastello F, Robert Y (2001) Matrix multiplication on heterogeneous platforms. IEEE Trans Parallel Distrib Syst 12(10):1033–1051. https://doi.org/10.1109/71.963416
2. Ben-Nun T, Hoefler T (2018) Demystifying parallel and distributed deep learning: an in-depth concurrency analysis. arXiv:1802.09941
3. Chen C, Weng Q, Wang W, Li B, Li B (2018) Fast distributed deep learning via worker-adaptive batch sizing. In: Proceedings of the ACM Symposium on Cloud Computing, SoCC ’18. ACM, New York, USA, pp 521–521
4. Chen J, Monga R, Bengio S, Jozefowicz R (2016) Revisiting distributed synchronous SGD. In: ICLR Workshop Track
5. Chiu C, Sainath TN, Wu Y, Prabhavalkar R, Nguyen P, Chen Z, Kannan A, Weiss RJ, Rao K, Gonina K, Jaitly N, Li B, Chorowski J, Bacchiani M (2017) State-of-the-art speech recognition with sequence-to-sequence models. arXiv:1712.01769
6. Clarke D, Zhong Z, Rychkov V, Lastovetsky A (2013) Fupermod: a framework for optimal data partitioning for parallel scientific applications on dedicated heterogeneous HPC platforms. In: Parallel Computing Technologies. Springer, Berlin, Heidelberg, pp 182–196
7. Dean J, Corrado GS, Monga R, Chen K, Devin M, Le QV, Mao MZ, Ranzato M, Senior A, Tucker P, Yang K, Ng AY (2012) Large scale distributed deep networks. In: NIPS, USA, pp 1223–1231
8. Forum MPI (2015) MPI: a message-passing interface standard, version 3.1 , June 4, 2015. High-Performance Computing Center Stuttgart, University of Stuttgart
9. Fox G, Qiu J, Jha S, Ekanayake S, Kamburugamuve S (2016) Big data, simulations and HPC convergence. In: Big Data Benchmarking. Springer, Cham, pp 3–17
10. Gupta S, Zhang W, Wang F (2017) Model accuracy and runtime tradeoff in distributed deep learning: a systematic study. In: IJCAI, pp 4854–4858
11. He K, Zhang X, Ren S, Sun J (2015) Deep residual learning for image recognition. arXiv:1512.03385
12. Hornik K (1991) Approximation capabilities of multilayer feedforward networks. Neural Netw 4(2):251–257
13. Huang Y, Cheng Y, Chen D, Lee H, Ngiam J, Le QV, Chen Z (2018) Gpipe: efficient training of giant neural networks using pipeline parallelism. arXiv:1811.06965
14. Jain AK, Mao J, Mohiuddin KM (1996) Artificial neural networks: a tutorial. Computer 29(3):31–44
15. Jiang J, Cui B, Zhang C, Yu L (2017) Heterogeneity-aware distributed parameter servers. In: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD ’17. ACM, NY, USA, pp 463–478
16. Krizhevsky A (2014) One weird trick for parallelizing convolutional neural networks. arXiv:1404.5997
17. Krizhevsky A, Sutskever I, Hinton GE (2012) Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems 25. Curran Associates, Inc., pp 1097–1105
18. Le QV, Ngiam J, Coates A, Lahiri A, Prochnow B, Ng AY (2011) On optimization methods for deep learning. In: Proceedings of the 28th International Conference on International Conference on Machine Learning, ICML’11. Omnipress, USA, pp 265–272
19. LeCun Y, Bengio Y, Hinton G (2015) Deep learning. Nature 521:436
20. Paoletti M, Haut J, Plaza J, Plaza A (2019) Deep learning classifiers for hyperspectral imaging: a review. ISPRS J Photogramm Remote Sens 158:279–317
21. Rico-Gallego JA, Díaz-Martín JC, Calvo-Jurado C, Moreno-Álvarez S, García-Zapata JL (2019) Analytical communication performance models as a metric in the partitioning of data-parallel kernels on heterogeneous platforms. J Supercomput 75(3):1654–1669
22. Schmidhuber J (2015) Deep learning in neural networks: an overview. Neural Netw 61:85–117
23. Sergeev A, Balso MD (2018) Horovod: fast and easy distributed deep learning in TensorFlow. arXiv:1802.05799

Affiliations

Sergio Moreno-Álvarez[1] · Juan M. Haut[2] · Mercedes E. Paoletti[2] · Juan A. Rico-Gallego[1] · Juan C. Díaz-Martín[2] · Javier Plaza[2]

Juan M. Haut
juanmariohaut@unex.es

Mercedes E. Paoletti
mpaoletti@unex.es

Juan A. Rico-Gallego
jarico@unex.es

Juan C. Díaz-Martín
juancarl@unex.es

Javier Plaza
jplaza@unex.es

[1] Department of Computer Systems Engineering and Telematics, University of Extremadura, Cáceres, Spain

[2] Department of Technology of Computers and Communications, University of Extremadura, Cáceres, Spain