Lab 5: Getting Started with Parallel Training of Neural Networks with TensorFlow

CC-MEI (FIB) Laboratory Assignment

Jordi Torres - Fall 2025

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

1. Hands-on exercise description

Deep learning models require significant computational resources, and training a model on a CPU can be extremely slow. Even with a single GPU, training large models can take hours or even days.

In this hands-on session, we will explore the impact of hardware acceleration by comparing CPU training vs. GPU training and then extending it to multi-GPU training using TensorFlow on a high-performance computing platform. Our goal is to answer the following questions:

- Why is training on a CPU inefficient?
- How much faster is a single GPU compared to a CPU?
- What happens when we increase the number of GPUs?

Important note: Beware of "copy & paste" as symbols such as commas or dashes can be incorrectly copied from this document to the console. If a command/code doesn't work correctly, write it directly!

Content

Laboratory practice instructions

Each hands-on exercise may include theoretical content to supplement lecture material, guiding students through the tasks that make up the practical work. Students are expected to read the instructions carefully before the corresponding laboratory session to ensure smooth progress during the hands-on activities.

Tasks should be completed in sequential order. However, while waiting for results (e.g., job execution), students may proceed with the next task if applicable.

It is strongly recommended to take digital notes of Intermediate steps performed (commands used, execution parameters), key results and observations or any encountered issues and attempted solutions.

Students are allowed to consult external sources and use AI-based tools (e.g., ChatGPT) to expand their understanding. However, it is crucial to critically evaluate AI-generated content, as these tools can produce errors or misleading information. If the submitted results contain errors due to an unchecked reliance on AI-generated outputs, this will significantly impact the evaluation of the hands-on assignment.

Submission Requirements: Once the hands-on session is completed, each group (or individual student) must submit their work to the designated inbox on the 'Racó de la FIB' intranet. The submission must include a slide-format presentation (PDF) detailing the results of each task, designed as if the audience had not done the exercise before.

2. Case Study

For this hands-on exercise, we have chosen a case study that involves classifying a set of images based on the popular CIFAR-10 dataset using the well-known ResNet family of neural networks, which is highly representative of how traditional neural networks are trained for computer vision.

This setup serves the pedagogical goals of this lab exercise, providing students with a practical introduction to parallel neural network training using TensorFlow. In this hands-on, all necessary code is provided in the following directory:

```
/gpfs/scratch/nct_XXX
├── MN5-NGC-TensorFlow-23.03.sif
├── ResNet50_seq.py
├── ResNet50_seq.CPU.slurm
├── ResNet50_seq.GPU.slurm
├── ResNet50.py
├── ResNet50.slurm
└── cifar-utils
    ├── __pycache__
    │   ├── cifar.cpython-37.pyc
    │   └── cifar.cpython-38.pyc
    ├── cifar-10-batches-py
    │   ├── batches.meta
    │   ├── data_batch_1
    │   ├── data_batch_2
    │   ├── data_batch_3
    │   ├── data_batch_4
    │   ├── data_batch_5
    │   └── test_batch
    └── cifar.py
```

To complete this hands-on, you need to copy the following files (code, not data) to your home directory and modify them accordingly. For example, make sure that the paths and the group specified in the Slurm files are correct (nct_XXX, as indicated by the professor in class).

Files to be copied locally:

```
ResNet50_seq.py
ResNet50_seq.CPU.slurm
ResNet50_seq.GPU.slurm
ResNet50.py
ResNet50.slurm
```

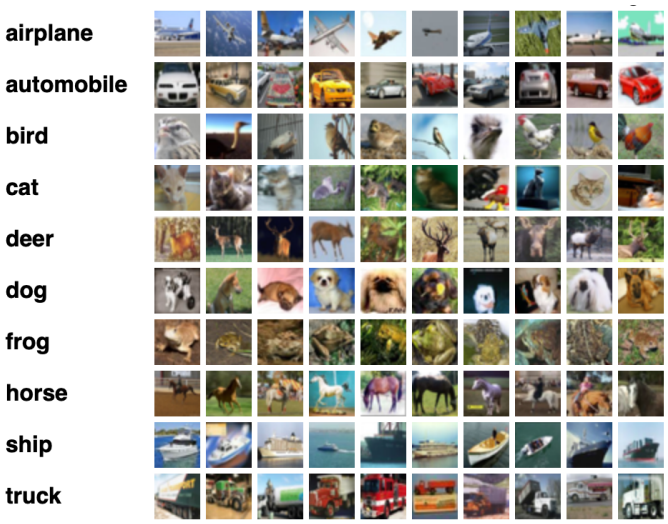
Let's take a quick look at the code and data for this case study.

Training data

CIFAR10 dataset

CIFAR-10 is a widely recognized dataset in computer vision, commonly used to introduce beginners to the task of object recognition. This dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. Compiled by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton, the dataset provides 50,000 images for training and 10,000 for testing (Learning Multiple Layers of Features from Tiny Images, Alex Krizhevsky[1], 2009).

Specifically, the dataset contains 50,000 training images and 10,000 test images. It is divided into five training batches and one test batch, each with 10,000 images. The test batch includes exactly 1,000 randomly selected images from each class. The training batches contain the remaining images in random order, with some batches potentially having more images from one class than another. Collectively, the training batches hold exactly 5,000 images per class. Here are the classes in the dataset, as well as ten random images from each[2]:



Dataset Preparation and Customization

We preloaded the CIFAR-10 dataset on the Marenostrum5 supercomputer[3]for this hands-on exercise. To increase the complexity of the training and allow for longer training times for better comparison, we have decided to apply a resizing operation to adjust the CIFAR-10 images to 128×128 pixels. To facilitate the experiments students will conduct, we have created a custom load_data[4] function that performs this resizing operation and splits the data into training and test sets.

---

[1] http://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[2] Image and text source: https://www.cs.toronto.edu/~kriz/cifar.html

[3] in the directory /gpfs/scratch/nct_XXX/cifar-utils/cifar-10-batches-py

[4] located in /gpfs/ scratch/nct_XXX/cifar-utils/load_cifar.py. *To use this custom function, we need to add the specific directory where the file is located. We do this with sys.path.append('<path>') in the Python code: sys.path.append('/gpfs/scratch/nct_XXX/cifar-utils')*

Model architectures

In deep learning, selecting an effective model architecture is essential for achieving strong performance, particularly in complex tasks like computer vision. Transfer learning, a common approach in this context, leverages pre-trained neural network architectures to improve training efficiency by building on previously learned patterns.

Transfer Learning

In the context of transfer learning, pre-trained neural network architectures are leveraged to improve training efficiency by building on previously learned patterns, which are especially useful in computer vision tasks. We will use a neural network with a specific architecture known as ResNet. In the scientific community, many prebuilt networks with established names are commonly used. For instance, AlexNet, developed by Alex Krizhevsky, is the architecture that won the ImageNet 2012 competition. GoogleLeNet, with its inception module, drastically reduces network parameters (15 times fewer than AlexNet). Others, such as VGGNet, demonstrated that network depth is a critical component for achieving strong results. The interesting thing about many of these networks is that they come preloaded in most deep learning frameworks. A list of available models for TensorFlow can be found in the Keras Application repository (where top-1 and top-5 accuracy refer to model performance on the ImageNet validation dataset).

Resnet

For this hands-on, we will consider one architecture from the ResNet family as a case study: ResNet50v2 neural network[5]. ResNet is a family of deep neural network architectures showing compelling accuracy and excellent convergence behaviors, introduced by He et al. in their 2015 paper, *Deep Residual Learning for Image Recognition*[6]. A few months later, the same authors published a new paper, *Identity Mapping in Deep Residual Network*[7], with a new proposal for the basic component, the residual unit, which makes training easier and improves generalization. And this lets the `V2` versions:

```
tf.keras.applications.ResNet50V2(
    include_top=True,
    weights="imagenet",
    input_tensor=None,
    input_shape=None,
    pooling=None,
    classes=1000,
    classifier_activation="softmax",
)
```

[5] https://keras.io/api/applications/resnet/#resnet50v2-function

[6] *https://arxiv.org/pdf/1512.03385.pdf*

[7] https://arxiv.org/pdf/1603.05027.pdf

The “50” stand for the number of weight layers in the network. The arguments for the network are:

- include_top: whether to include the fully connected layer at the top of the network.
- weights: one of `None` (random initialization), ‘`imagenet`' (pre-training on ImageNet), or the path to the weights file to be loaded.
- input_tensor: optional Keras tensor (i.e. the output of `layers.Input()`) to use as image input for the model.
- input_shape: optional shape tuple, only to be specified if `include_top` is False (otherwise, the input shape has to be `(224, 224, 3)` (with `‘channels_last’` data format) or `(3, 224, 224)` (with `‘channels_first’` data format). It should have exactly 3 input channels, and the width and height should be no smaller than 32. E.g. `(200, 200, 3)` would be one valid value.
- pooling: Optional pooling mode for feature extraction when `include_top` is `False`. (a) `None` means that the output of the model will be the 4D tensor output of the last convolutional block. (b) `avg` means that global average pooling will be applied to the output of the previous convolutional block, and thus, the output of the model will be a 2D tensor. (c) `max` means that global max pooling will be applied.
- classes: optional number of classes to classify images into, only to be specified if `include_top` is `True`, and if no `weights` argument is specified.
- classifier_activation: A `str` or callable. The activation function to use on the “top” layer. Ignored unless `include_top=True`. Set `classifier_activation=None` to return the logits of the “top” layer.

Note that if `weights="imagenet"`, Tensorflow middleware requires a connection to the internet to download the imagenet weights (pre-training on ImageNet). Due we are not centering our interest in Accuracy, we didn’t download the file with the imagenet weights; therefore, it must be used `weights=None`.

Basic TensorFlow Code for Sequential Training

Let’s start with a sequential version of the training in order to familiarize ourselves with the classifier. The sequential code to train the previously described problem of classification of the CIFAR10 dataset using a ResNet50 neural network could be the following:

```
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras import models
import numpy as np
import argparse
import time
import sys
```

```
sys.path.append('/gpfs/scratch/nct_XXX/cifar-utils')
from cifar import load_cifar

parser = argparse.ArgumentParser()
parser.add_argument('--epochs', type=int, default=5)
parser.add_argument('--batch_size', type=int, default=2048)
args = parser.parse_args()

batch_size = args.batch_size
epochs = args.epochs

train_ds, test_ds = load_cifar(batch_size)

model = tf.keras.applications.resnet_v2.ResNet50V2(
        include_top=True,
        weights=None,
        input_shape=(128, 128, 3), classes=10)

opt = tf.keras.optimizers.SGD(0.01)
model.compile(loss='sparse_categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])
model.fit(train_ds, epochs=epochs, verbose=2)
```

Task 1: Code Review and Understanding (10%)

Please carefully review the attached code to understand its behavior fully. If you have any questions, feel free to contact the instructor. Prepare the answer including this code and a brief explanation of the main parts of it.

---

3. Sequential Training vs. Basic Parallel Training

Let's start training using only CPUs and observe that training times, even for a small but realistic network, become quite lengthy, confirming that training a modern neural network without GPUs is impractical. To demonstrate this, we will run a single epoch using only CPUs and then another using a GPU. With just two epoch, we can compare and see GPUs' importance.

In summary, we will explore the impact of hardware acceleration by comparing CPU training vs. GPU training.

Train on CPU

To determine how long it takes to train a model on a CPU, we can use the following Slurm script, located in `/gpfs/scratch/nct_XXX`. This script (`ResNet50_seq.CPU.slurm`) is designed to run the `ResNet50_seq.py` code using only a CPU[8]:

```
#!/bin/bash
#SBATCH --chdir .
#SBATCH --job-name=ResNet50_seq_CPU
#SBATCH --output=%x.%j.out
#SBATCH --error=%x.%j.err
#SBATCH --nodes 1
#SBATCH --ntasks-per-node 1
#SBATCH --cpus-per-task 20
#SBATCH --time 01:15:00
#SBATCH --account nct_XXX
#SBATCH --qos acc_debug

module purge
module load singularity

SINGULARITY_CONTAINER=/gpfs/scratch/nct_XXX/MN5-NGC-TensorFlow-
23.03.sif
singularity exec $SINGULARITY_CONTAINER python ResNet50_seq.py --
epochs 2 --batch_size 256
```

Task 2: Training on CPU (Baseline Performance) (10%)

Run the script and obtain the execution results from the standard output file. How long did the first epoch take? How long did the second one take? Why is there a difference?

Train on a Single GPU

To run the same python code using the SLURM system that allocates a GPU requires adding the following flag in the slurm file

```
#SBATCH --gres=gpu:1
```

[8] Singularity container is used in the Slurm file to execute the Python code.

Also, the `--nv` flag is required in the `singularity exec` command to enable GPU access to the container. It automatically mounts the necessary NVIDIA libraries (such as CUDA and cuDNN) within the container, allowing applications inside the container to use the system's GPUs.

The following Slurm file, located in `/gpfs/scratch/nct_XXX`, can be used to run the `ResNet50_seq.py` code using a GPU (`ResNet50_seq.GPU.slurm`):

```
#!/bin/bash
#SBATCH --chdir .
#SBATCH --job-name=ResNet50_seq_GPU
#SBATCH --output=%x.%j.out
#SBATCH --error=%x.%j.err
#SBATCH --nodes 1
#SBATCH --ntasks-per-node 1
#SBATCH --cpus-per-task 20       #Minimum cpus requested should
                                 # be (nodes * gpus/node * 20)
#SBATCH --time 01:15:00
#SBATCH --account nct_XXX
#SBATCH --qos acc_debug

#SBATCH --gres gpu:1

echo " "
echo "Job Name: $SLURM_JOB_NAME"
echo "Job ID: $SLURM_JOB_ID"
echo "Submit Directory: $SLURM_SUBMIT_DIR"
echo "Partition: $SLURM_JOB_PARTITION"
echo "Account: $SLURM_JOB_ACCOUNT"
echo " "
echo "Number of Nodes Allocated: $SLURM_JOB_NUM_NODES"
echo "Total number of tasks: $SLURM_NTASKS"
echo "Number of tasks per node: $SLURM_NTASKS_PER_NODE"
echo "Number of CPU cores per task: $SLURM_CPUS_PER_TASK"
echo " "
echo "Specific GPUs allocated:"
nvidia-smi

module purge
module load singularity

SINGULARITY_CONTAINER=/gpfs/scratch/nct_XXX/MN5-NGC-TensorFlow-
23.03.sif
singularity exec --nv $SINGULARITY_CONTAINER python ResNet50_seq.py
--epochs 1 --batch_size 256
```

We see that this Slurm script includes several `echo` commands that will write to the standard output file the resources Slurm has allocated, allowing us to verify them if a detailed analysis of the results is needed.

We also use `nvidia-smi`, which alone provides a quick summary of all GPUs in the system, enabling us to check if the GPUs specified by the corresponding flag are indeed being allocated.

Task 3: Comparing Execution on CPU vs. GPU (10%)

What was the execution time of the same program when run on a GPU? Discuss the differences in execution time between the CPU and GPU in your answer. Include any additional information you consider relevant.

Why just two epoch? While it's true that the achieved accuracy is poor and increasing the number of epochs would improve it, this would require additional time and resources that aren't necessary to spend, given that the academic purpose of this hands-on exercise focuses on execution efficiency rather than algorithmic performance.

4. Accelerate the Learning with Parallel Training

In HPC, distributed training is essential for scaling deep learning to larger models and datasets. By splitting training across multiple GPUs, it's possible to handle workloads that would be impractical on a single GPU, both in terms of time and memory.

Now that we have explored single-device training (CPU vs. single GPU), we move to the next step: leveraging multiple GPUs to accelerate training. In this phase, we will introduce TensorFlow's `tf.distribute.MirroredStrategy`, which enables data-parallel training across multiple GPUs.

Basic Concepts

Performance Metrics: Speedup, Throughput, and Scalability

To accelerate the training process, we need performance metrics to measure it. In these systems, the term "performance"has a dual meaning. On one hand, it refers to the predictive accuracy of the model. On the other, it refers to the computational speed of the process. Accuracy is independent of computational resources and serves as the key performance metric for comparing different deep neural network models.

In contrast, computational speed depends on the platform on which the model is deployed. We will measure this using metrics such as Speedup, which is the ratio of the execution time of the sequential algorithm (using a single GPU in our hands-on exercises) to the execution time of the parallel algorithm (using multiple GPUs). In other words, it is the ratio between the time taken for sequential execution (1 GPU) and parallel execution (N GPUs). Example: If training one epoch with 1 GPU takes 80 seconds, and with 4 GPUs it takes 25 seconds, then: Speedup=80/25=3.2×. An ideal linear speedup would be 4×, meaning that 3.2× indicates some overhead.

Another important metric is Throughput, which, in general terms, refers to the rate of production or the speed at which something is processed. For example, it can be measured as the number of images processed per unit of time. Throughput provides a useful benchmark for computational performance, though it can vary depending on the type of neural network being used.

Finally, we have Scalability, a broader concept that refers to a system's ability to efficiently handle increasing amounts of work. Scalability heavily depends on factors such as the configuration of the computing cluster, the type of network architecture, and the efficiency of the deep learning framework in utilizing libraries and managing resources. It is a qualitative metric that helps us assess whether increasing the number of GPUs continues to provide proportional benefits. Example: When scaling from 1 to 2 GPUs, we might achieve excellent efficiency (~90%). However, when scaling from 2 to 4 GPUs, efficiency might drop (~80%, as in the previous example).

These metrics will be applied in the following hands-on exercises to help interpret the results. Throughput (measured in images per second) and Speedup (relative training time improvement) are key metrics for understanding the effectiveness of parallel training. Higher throughput indicates greater efficiency, while speedup helps evaluate whether adding more GPUs leads to proportional benefits.

Types of parallelism

The parallel and distributed training approach is widely used by deep learning practitioners. This is because today's neural networks are compute-intensive, making them comparable to traditional high-performance computing (HPC) applications. As a result, large learning workloads perform exceptionally well on accelerated systems like clusters of general-purpose graphics processing units (GPUs), which have long been used in the supercomputing field.

There are many implementations to achieve distributed training, and the choice between them will depend on the application's specific needs. In some cases, a combination of both approaches can enhance performance.

One approach is Model Parallelism, where different layers of a deep learning model are trained in parallel across multiple GPUs. This method allows the distribution of model parameters, which is particularly useful when the model is too large to fit into the memory of a single GPU.

Another widely used approach is Data Parallelism, where the same model is replicated on each computing device, but each device trains the model using different subsets of the data. In this case, each GPU processes its batch of data in parallel, and after computing the gradients, the results are synchronized across all devices.

In addition to these two main methods, there are other advanced techniques to optimize further the training of large language models, such as Tensor Parallelism, Pipeline Parallelism, and Sequence Parallelism, which are crucial for training models at a massive scale using thousands of GPUs. When combined with model and data parallelism, these techniques can significantly boost performance and minimize training time. However, all these techniques are beyond the scope of this course due to their complexity.

In this hands-on session, we will focus on data parallelism, an approach that enables efficient scaling across multiple GPUs and represents the simplest form of parallelism in neural network training. Fortunately, deep learning (DL) frameworks facilitate this kind of parallelization or distribution.

Task 4: Performance metrics and types of parallelism (10%)

Prepare a slide explaining the key performance metrics (Speedup, Throughput, and Scalability) and another slide describing the main types of parallelism (Model Parallelism and Data Parallelism). Clearly define each concept and its relevance in deep learning training, as these metrics and techniques are essential for optimizing performance in multi-GPU environments.

Parallel training with TensorFlow

We will use the TensorFlow framework to parallelize training on a server with 4 GPUs, simplifying data parallelism with the `tf.distribute.Strategy` API, which enables DL model training by distributing datasets across multiple devices.

TensorFlow for multiple GPUs

`tf.distribute.Strategy` is a TensorFlow API to distribute training across multiple GPUs (or TPUs) with minimal code changes (from the sequential version). This API can be used with a high-level API like Keras and can also be used to distribute custom training loops.

`tf.distribute.Strategy` intends to cover a number of distribution strategies use cases along different axes. The official web page[9] of this feature presents all the currently supported combinations; however, in this hands-on, we will focus our attention on `tf.distribute.MirroredStrategy` is one of the strategies included in `tf.distribute.Strategy`.

"MirroredStrategy"

We will use `tf.distribute.MirroredStrategy` , in this hands-on exercise, that supports the training process on multiple GPUs (multiple devices) on one server (single host). It creates one replica per GPU device. Each variable in the model is mirrored across all the replicas. These variables are kept in sync with each other by applying identical updates.

Let's assume we are on a single machine that has multiple GPUs, and we want to use more than one GPU for training. We can accomplish this by creating our `MirroredStrategy`:

```
mirrored_strategy = tf.distribute.MirroredStrategy()
```

This will create a `MirroredStrategy` instance that will use all the GPUs visible to TensorFlow. It is possible to see the list of available GPU devices doing the following:

```
devices = tf.config.experimental.list_physical_devices("GPU")
```

It is also possible to use a subset of the available GPUs in the system by doing the following[10]:

```
mirrored_strategy =
tf.distribute.MirroredStrategy(devices=["/gpu:0", "/gpu:1"])
```

We then need to declare our model architecture and compile it within the scope of the `MirroredStrategy`. To build the model and compile it inside the `MirroredStrategy` scope we can do it in the following way:

```
with mirrored_strategy.scope():
     model = tf.keras.applications.resnet_v2.ResNet50V2(
             include_top=True, weights=None,
             input_shape=(128, 128, 3), classes=10)

     opt = tf.keras.optimizers.SGD(learning_rate)

     model.compile(loss='sparse_categorical_crossentropy',
```

[9] https://www.tensorflow.org/guide/distributed_training

[10] Newer versions of TensorFlow (2.10+) have improved the distributed training API, making it easier to use. For our case of multi-GPU training on a single node, tf.distribute.MirroredStrategy remains the recommended option, but it is now even simpler to use with Keras. For example, to list available devices, we can directly use tf.config.list_physical_devices('GPU')

```
                optimizer=opt, metrics=['accuracy'])
```

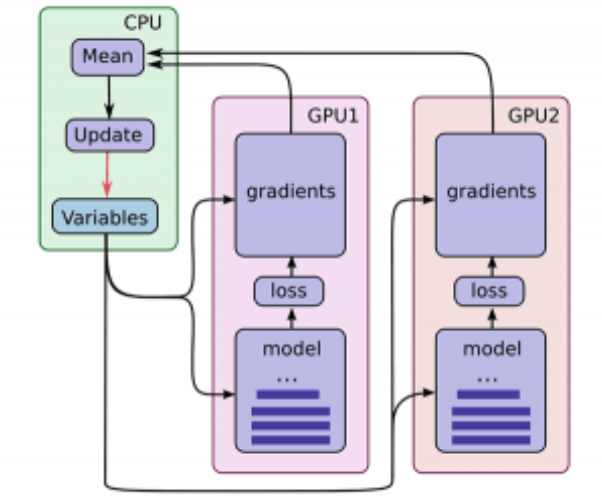
This allows us to create distributed variables instead of regular variables: each variable is mirrored across all the replicas and is kept in sync with each other by applying identical updates. It is important during the coding phase that the creation of variables should be under the strategy scope. In general, this is only during the model construction step and the compile step. Training can be done as usual outside the strategy scope with:

```
dataset = load_data(batch_size)
model.fit(dataset, epochs=5, verbose=2)
```

Centralized synchronous distributed training

In centralized synchronous distributed training, `tf.distribute.MirroredStrategy` ensures that the state of model replicas across multiple GPUs remains synchronized at all times. This means that each GPU has an identical copy of the model, and during training, all GPUs will compute gradients independently but apply them collectively, ensuring consistent model updates.

When a `MirroredStrategy` scope is opened, and a model is created within this scope, the `MirroredStrategy` object automatically generates one model replica on each available GPU. Each replica processes a portion of the input data independently, allowing training to happen in parallel. However, the model variables (such as weights) are mirrored across all replicas, keeping them consistent after each update.



The image above (image source[11]) illustrates this process in a setup with two GPUs (GPU1 and GPU2) and a central CPU for managing model updates:

[11] https://jhui.github.io/2017/03/07/TensorFlow-GPU/

1. Model Replication: On the left side, we see a central CPU managing the variables and updates. Each GPU (GPU1 and GPU2) has a copy of the model, enabling parallel computation across GPUs.
2. Forward Pass and Gradient Calculation: Each GPU processes its own data and computes the loss independently. The model on each GPU calculates gradients based on its portion of the data. This parallel computation reduces the overall training time.
3. Gradient Aggregation and Update: The gradients from each GPU are sent to the CPU, which averages them. This averaged gradient is then used to update the model variables. Once updated, these variables are synchronized back to each GPU, ensuring that all model replicas maintain the same state. This process of averaging gradients and synchronizing variables ensures that each model replica remains identical, allowing the training to proceed in a centrally synchronized manner.

Task 5: Parallel Training with TensorFlow (10%)

Prepare a slide explaining how TensorFlow's tf.distribute.MirroredStrategy enables parallel training across multiple GPUs. Describe how this strategy replicates the model across GPUs, keeps variables synchronized, and performs centralized synchronous distributed training. Include key code snippets demonstrating how to initialize and use MirroredStrategy to distribute training efficiently.

Multi-GPU Training with MirroredStrategy

Parallel performance measurement

Remember that in this hands-on exercise, we will consider epoch time as a measure of the computation time needed to train a distributed neural network. This approximated measure in seconds, provided by Keras using the `fit` method, is accurate enough for the purpose of this academic exercise. In our case, we suggest discarding the first-time epoch, which includes creating and initializing structures. Obviously, for certain types of performance studies, it is necessary to go into more detail, differentiating the loading data, feed-forward time, loss function time, backpropagation time[12], etc., but it falls outside the scope of this case study we propose in this hands-on exercise.

[12] https://arxiv.org/pdf/1909.02061

Batch Size and Learning Rate

When training, memory allocation is required to store samples for training the model as well as the model itself. We need to keep this in mind to avoid an out-of-memory error.

Remember that `batch_size` is the number of samples the model will process at each training step, and generally, we aim to make this number as large as possible. One way to determine an optimal `batch_size` is by trial and error, testing different values until a memory capacity error occurs. For now, in this introductory practice on neural network training, we won't calculate the batch_size; instead, we will set it to 2048 for our single GPU executions.

When using `MirroredStrategy` with multiple GPUs, the batch size indicated is divided by the number of replicas. Therefore, the `batch_size` that we should specify to TensorFlow is equal to the maximum value for one GPU multiplied by the number of GPUs we are using. This is, in our example, use these flags in the python program:

```
python ResNet50.py -- epochs 5 -- batch_size 2448 -- n_gpus 1
python ResNet50.py -- epochs 5 -- batch_size 4096 -- n_gpus 2
python ResNet50.py -- epochs 5 -- batch_size 8192 -- n_gpus 4
```

Accordingly, with the `batch_size`, if we are using `MirroredStrategy` with multiple GPUs, we change the `learning_rate` to `learning_rate*num_GPUs`:

```
learning_rate = learning_rate_base*number_of_gpus
opt = tf.keras.optimizers.SGD(learning_rate)
```

Parallelization of the training step

In this section, we will show how we can parallelize the training step on the Marenostrum 5 ACC cluster.

Parallel code for ResNet50 neural network

Following the steps presented in the above section on how to apply `MirroredStrategy`, below we present the resulting parallel code for the ResNet50 (`ResNet50.py`):

```
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras import models
import numpy as np
import argparse
import time
import sys
sys.path.append('/gpfs/scratch/nct_XXX/cifar-utils')
from cifar import load_cifar
parser = argparse.ArgumentParser()
parser.add_argument(' -- epochs', type=int, default=5)
parser.add_argument(' -- batch_size', type=int, default=2048)
parser.add_argument(' -- n_gpus', type=int, default=1)
args = parser.parse_args()
```

```
batch_size = args.batch_size
epochs = args.epochs
n_gpus = args.n_gpus
train_ds, test_ds = load_cifar(batch_size)
device_type = 'GPU'
devices = tf.config.experimental.list_physical_devices(
          device_type)
devices_names = [d.name.split("e:")[1] for d in devices]
strategy = tf.distribute.MirroredStrategy(
           devices=devices_names[:n_gpus])
with strategy.scope():
     model = tf.keras.applications.resnet_v2.ResNet50V2(
             include_top=True, weights=None,
             input_shape=(128, 128, 3), classes=10)
     opt = tf.keras.optimizers.SGD(0.01*n_gpus)
     model.compile(loss='sparse_categorical_crossentropy',
                   optimizer=opt, metrics=['accuracy'])
model.fit(train_ds, epochs=epochs, verbose=2)
```

SLURM script

The SLURM script that allocates resources and executes the model for different numbers of GPUs can be as the following one (`ResNet50.slurm`):

```
#!/bin/bash
#SBATCH --chdir .
#SBATCH --job-name=ResNet50
#SBATCH --output=%x.%j.out
#SBATCH --error=%x.%j.err
#SBATCH --nodes 1
#SBATCH --ntasks-per-node 1
#SBATCH --gres gpu:4
#SBATCH --cpus-per-task 80
#SBATCH --time 01:15:00
#SBATCH --account nct_XXX
#SBATCH --qos acc_debug

echo " "
echo "Job Name: $SLURM_JOB_NAME"
echo "Job ID: $SLURM_JOB_ID"
echo "Submit Directory: $SLURM_SUBMIT_DIR"
echo "Partition: $SLURM_JOB_PARTITION"
echo "Account: $SLURM_JOB_ACCOUNT"
echo " "
```

```
echo "Number of Nodes Allocated: $SLURM_JOB_NUM_NODES"
echo "Total number of tasks: $SLURM_NTASKS"
echo "Number of tasks per node: $SLURM_NTASKS_PER_NODE"
echo "Number of CPU cores per task: $SLURM_CPUS_PER_TASK"
echo " "
echo "Specific GPUs allocated:"
nvidia-smi

module purge
module load singularity

SINGULARITY_CONTAINER=/gpfs/scratch/nct_XXX/MN5-NGC-TensorFlow-
23.03.sif
singularity exec --nv $SINGULARITY_CONTAINER python ResNet50.py --
epochs 5 --batch_size 2048 --n_gpus 1
singularity exec --nv $SINGULARITY_CONTAINER python ResNet50.py --
epochs 5 --batch_size 4096 --n_gpus 2
singularity exec --nv $SINGULARITY_CONTAINER python ResNet50.py --
epochs 5 --batch_size 8192 --n_gpus 4
```

It is important to note that we center our interest on the computational speed of the process rather than the model's accuracy. For this reason, we will only execute a few epochs during the training to compare the three options. It is obvious that if we want to do a deep/real performance analysis, we should carry out several tests and then take the averages of the times obtained in each one of them. However, given the academic purpose of this exercise (and the cost of supercomputing resources offered by BSC, which we must save!), it is enough for only one execution.

Task 6: Parallelization of ResNet50 (30%)

Run the parallel `ResNet50.py` program on the Marenostrum5 ACC cluster using 1, 2, and 4 GPUs.

1. **Plot the epoch time (in seconds) against the number of GPUs in a bar chart. Given that we know there are 50,000 images, we can convert this to images per second, representing the throughput.**
2. **Plot the throughput (images per second) vs. the number of GPUs. How does throughput change as the number of GPUs increases?**
3. **Since speedup is the most relevant metric, plot it in a bar chart as well. Is the speedup nearly linear? Recall that we consider speedup to be linear when the workload is evenly distributed across GPUs.**

Analyze the results and explain your findings.

Note that in the file containing the standard output, nvidia-smi now indicates that we are using all 4 GPUs. Remember that the output of nvidia-smi provides detailed information about the status of the NVIDIA GPUs available in the system. It includes the driver and CUDA versions, the temperature, power consumption, and memory usage of each GPU, as well as its utilization percentage. Additionally, it displays any active processes running on the GPUs. In this case, when the nvidia-smi command was executed within the Slurm script, the training process had already finished. As a result, the GPUs appear unoccupied, showing 0% utilization and minimal memory usage. This indicates that no active processes were running at the time of the query, which is expected once the execution has completed.

5. Scaling Deep Networks and Speedup Limitations

In previous section we successfully scaled training across multiple GPUs using MirroredStrategy, but we observed that speedup was not always ideal. Now, in this section, we will take a deeper look at scaling deep neural networks, particularly training larger models (ResNet152) and investigating why speedup is not perfectly linear.

ResNet152

We want to highlight that the size of the network, in terms of the number of parameters, for example, directly impacts the system's resource requirements. Without going into too much detail regarding hyperparameter tunning, we propose comparing the neural network we've already used, `resnet_v2.ResNet50V2`, with `resnet_v2.ResNet152V2`, which has more than twice the number of parameters.

It is important to note that we are omitting many key details about hyperparameters that need to be fine-tuned for optimal model accuracy during training. However, this is not the focus of this hands-on exercise, as it would require significant resources beyond our availability. Instead, this approach provides a clear perspective on the impact of hardware resource requirements in each case.

We can use the same code presented in previous sections to train any other networks available in Keras[13]. The only adjustment needed is to replace the network name with the desired network in the program. This is one of the advantages of transfer learning.

As mentioned, for the purpose of this lab, the metric we will use to compare performance is the execution time, which can be tracked by the time Keras itself reports for each epoch. However, in general, we discard the first epoch as it differs from the rest due to the creation and initialization of memory structures. For this reason, in this section, we will run five epochs for each Slurm test (now using the GPU).

[13] https://keras.io/api/applications/

Parallelization of the ResNet152V2 neural network

Now it's your turn to get your hands really dirty and reproduce the above results for a new neural network, the `ResNET152V2` classifier.

Remember that we aim to make `batch_size` as large as possible. One way to determine an optimal `batch_size` is by trial and error, testing different values until a memory capacity error occurs. In the previous example we won't calculate the batch_size; instead, we set it to 2048. Check if the batch size needs to be reduced with this new network. You can verify it if you encounter a memory error. In that case, halve the batch size.

Task 7: Parallelization of ResNet152 (10%)

Create a new `.py` file and a new `.slurm` script to run parallelization experiments (for 1,2 and 4 GPUs) with the `ResNet152V2` classifier. Explain your findings.

Task 8: Comparing ResNet50 vs ResNet152 (10%)

Plot the results of both case studies together in a bar chart to highlight key differences. First, observe that the epoch time for ResNet152 is significantly longer, resulting in much lower throughput (images per second) compared to the ResNet50 network. Why might this be the case? Could it be that ResNet152, being a deeper network with more layers, naturally increases training time?

Compare the speedup values between ResNet50 and ResNet152; Do larger models experience slower or less consistent speedup? Also, notice that the speedup for ResNet152 is no longer linear. Why might this happen? Could one likely factor be that the larger network size introduces additional latency for synchronization, impacting performance?

Prepare the necessary answer with the plots to present your findings.

---

6. Acknowledgment

Many thanks to Juan Luis Domínguez and Oriol Aranda, who wrote the first version of the codes that appear in this hands-on, and to Carlos Tripiana and Félix Ramos for their essential support. Also, many thanks to Alvaro Jover Alvarez, Miquel Escobar Castells, and Raul Garcia Fuentes for their contributions to proofreading the initial versions of this document.