Tarantella Tutorial

Release 1.0

Alexandra Carpen-Amarie

Sep 21, 2022

CONTENTS

Tarantella is an open-source, distributed Deep Learning framework built on top of TensorFlow, providing scalable Deep Neural Network training on CPU and GPU compute clusters.

Tarantella is easy-to-use, allows to re-use existing TensorFlow models, and does not require any knowledge of parallel computing.

Distributed training in Tarantella is based on the simplest and most efficient parallelization strategy for deep neural networks (DNNs), which is called *data parallelism*.

This strategy is already in use when deploying batched optimizers, such as stochastic gradient descent (SGD) or ADAM. In this case, input samples are grouped together in so-called mini-batches and are processed in parallel.

Tarantella extends this scheme by splitting each mini-batch into a number of micro-batches, which are then executed on different devices (e.g., GPUs). In order to do this, the DNN is replicated on each device, which then processes part of the data independently of the other devices. During the *backpropagation* pass, partial results need to be accumulated via a so-called allreduce collective operation.

CHAPTER

ONE

INSTALLATION

Tarantella needs to be built from source. Since Tarantella is built on top of TensorFlow, you will require a recent version of it. Additionally, you will need an installation of the open-source communication libraries GaspiCxx and GPI-2, which Tarantella uses to implement distributed training.

Lastly, you will need pybind11, which is required for Python and C++ inter-communication.

In the following we will look at the required steps in detail.

1.1 Installing dependencies

1.1.1 Compiler and build system

Tarantella can be built using a recent gcc compiler with support for C++17 (starting with `gcc 7.4.0`). You will also need the build tool CMake (from version `3.12`).

1.1.2 Installing TensorFlow

First you will need to install TensorFlow. Supported versions range between `Tensorflow 2.4 - 2.9`, and they can be installed in a conda environment using pip, as recommended on the TensorFlow website.

> **Caution:** This tutorial targets installations on the STYX cluster, where some of the dependencies are pre-installed. For a full description of Tarantella's installation steps, refer to the Tarantella documentation.

To get started, create and activate an environment for Tarantella:

```
conda create -n tarantella
conda activate tarantella
```

Now, you can install TensorFlow with:

```
conda install -c nvidia python==3.9 cudatoolkit~=11.2 cudnn
pip install --upgrade tensorflow_gpu==2.9
conda install pybind11 pytest networkx
```

Tarantella requires at least Python `3.7`. Make sure the selected version also matches the TensorFlow requirements.

Caution: If a new *conda* environment is created, or Tensorflow is reinstalled/updated, please repeat all the steps concerning the *Tarantella library installation*. The Tarantella backend is compiled against the TensorFlow library, and thus it requires exactly the same binary version at runtime.

1.1.3 Installing GaspiCxx

GaspiCxx is a C++ abstraction layer built on top of the GPI-2 library, designed to provide easy-to-use point-to-point and collective communication primitives. Tarantella's communication layer is based on GaspiCxx and its PyGPI API for Python.

To install GaspiCxx and PyGPI, first download the latest `dev` branch from the git repository:

```
git clone https://github.com/cc-hpc-itwm/GaspiCxx.git
cd GaspiCxx
git checkout dev
```

Compile and install the library as follows, making sure the previously created conda environment is activated:

```
conda activate tarantella

mkdir build && cd build
export GASPICXX_INSTALLATION_PATH=/your/gaspicxx/installation/path
cmake -DBUILD_PYTHON_BINDINGS=ON     \
      -DBUILD_SHARED_LIBS=ON         \
      -DCMAKE_INSTALL_PREFIX=${GASPICXX_INSTALLATION_PATH} ../
make -j$(nproc) install
```

where `${GASPICXX_INSTALLATION_PATH}` needs to be set to the path where you want to install the library.

1.2 Building Tarantella from source

With all dependencies installed, we can now download, configure and compile Tarantella. To download the source code, simply clone the GitHub repository:

```
git clone https://github.com/cc-hpc-itwm/tarantella.git
cd tarantella
git checkout master
```

Next, we need to configure the build system using CMake. For a standard out-of-source build, we create a separate `build` folder and run `cmake` in it:

```
conda activate tarantella

cd tarantella
mkdir build && cd build
export TARANTELLA_INSTALLATION_PATH=/your/installation/path
cmake -DCMAKE_INSTALL_PREFIX=${TARANTELLA_INSTALLATION_PATH} \
      -DCMAKE_PREFIX_PATH=${GASPICXX_INSTALLATION_PATH} ../
```

Now, we can compile and install Tarantella to `TARANTELLA_INSTALLATION_PATH`:

```
make -j$(nproc) install
export PATH=${TARANTELLA_INSTALLATION_PATH}/bin:${PATH}
```

1.3 [Optional] Building and running tests

In order to build Tarantella with tests, please follow the steps from the Tarantella docs.

CHAPTER

TWO

QUICK START

This section explains how to start using Tarantella to distributedly train an existing TensorFlow model.

Note: Tarantella is composed of two different components that need to be used together for data parallel training across multiple devices.

1. Python module that can be imported in your code and provides access to the Tarantella API.
2. Runtime execution script *tarantella* to deploy the code in parallel.

Now, we will examine what changes have to be made to your code, and how to execute it on the command line with `tarantella`.

2.1 Code example: LeNet-5 on MNIST

After having *built and installed* Tarantella we are ready to add distributed training support to an existing TensorFlow model. We will first illustrate all the necessary steps, using the well-known example of **LeNet-5** on the **MNIST** dataset. Although this is not necessarily a good use case to take full advantage of Tarantella's capabilities, it will allow you to simply copy-paste the code snippets and try them out, even on your laptop.

Let's get started!

```
import tensorflow as tf
from tensorflow import keras

# Initialize Tarantella (before doing anything else)
import tarantella as tnt

# Skip function implementations for brevity
[...]

args = parse_args()

# Create Tarantella model from a `keras.Model`
model = tnt.Model(lenet5_model_generator())

# Compile Tarantella model (as with Keras)
model.compile(optimizer = keras.optimizers.SGD(learning_rate=args.learning_rate),
              loss = keras.losses.SparseCategoricalCrossentropy(),
              metrics = [keras.metrics.SparseCategoricalAccuracy()])
```

(continues on next page)

(continued from previous page)

```

# Load MNIST dataset (as with Keras)
shuffle_seed = 42
(x_train, y_train), (x_val, y_val), (x_test, y_test) = \
      mnist_as_np_arrays(args.train_size, args.val_size, args.test_size)

train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
train_dataset = train_dataset.shuffle(len(x_train), shuffle_seed)
train_dataset = train_dataset.batch(args.batch_size)
train_dataset = train_dataset.prefetch(tf.data.experimental.AUTOTUNE)

test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test))
test_dataset = test_dataset.batch(args.batch_size)

# Train Tarantella model (as with Keras)
model.fit(train_dataset,
          epochs = args.number_epochs,
          verbose = 1)

# Evaluate Tarantella model (as with Keras)
model.evaluate(test_dataset, verbose = 1)
```

As you can see from the marked lines in the code snippet, you only need to add *two lines of code* to train LeNet-5 distributedly using Tarantella! Let us go through the code in some more detail, in order to understand what is going on.

First we need to import the Tarantella library:

```
import tarantella as tnt
```

Importing the Tarantella package will initialize the library and set up the communication infrastructure. Note that this should be done before executing any other code.

Next, we need to wrap the `keras.Model` object, generated by `lenet5_model_generator()`, into a `tnt.Model` object:

```
model = tnt.Model(lenet5_model_generator())
```

That's it!

All the necessary steps to distribute training and datasets will now be automatically handled by Tarantella. In particular, we still run `model.compile` on the new `model` to generate a compute graph, just as we would have done with a typical Keras model.

Next, we load the MNIST data for training and testing, and create `tf.data.Dataset` s from it. Note that we `batch` the dataset for training. This will guarantee that Tarantella is able to distribute the data later on in the correct way. Also note that the `batch_size` used here, is the same as for the original model, that is the *global* batch size. For details concerning local and global batch sizes have a look here.

Now we are able to train our `model` using `model.fit`, in the same familiar way used by the standard Keras interface. Note, however, that Tarantella is taking care of the proper distribution of the `train_dataset` in the background. All the possibilities of how to feed datasets to Tarantella are explained in more detail below. Lastly, we can evaluate the final accuracy of our `model` on the `test_dataset` using `model.evaluate`.

To test and run Tarantella in the next section, you can find a full version of the above example here.

2.2 Executing your model with `tarantella`

Next, let's execute our model distributedly using `tarantella` on the command line.

Caution: When working on **STYX**, make sure to export the following environment variables before calling *tarantella*:

```
export LD_LIBRARY_PATH=/opt/GPI/lib64:${LD_LIBRARY_PATH}
export LD_LIBRARY_PATH=${GASPICXX_INSTALLATION_PATH}:${LD_LIBRARY_PATH}
```

The simplest way to run the model is by passing its Python script to `tarantella`:

```
tarantella -- model.py
```

This will execute our model distributedly on a single node, using all the available GPUs.

Caution: On **STYX**, you might run into some error messages when trying to use the GPUs. Follow the following steps to correctly run Tarantella:

```
export CONDA_ENV_PATH=/path/to/your/conda/environment

mkdir -p ${CONDA_ENV_PATH}/lib/nvvm/libdevice
mv ${CONDA_ENV_PATH}/lib/libdevice.10.bc ${CONDA_ENV_PATH}/lib/nvvm/libdevice
export LD_LIBRARY_PATH=${CONDA_ENV_PATH}/lib:${LD_LIBRARY_PATH}
```

Always add the following `-x` flags to the `tarantella` command in the examples bellow:

```
tarantella -x XLA_FLAGS="--xla_gpu_cuda_data_dir=${CONDA_ENV_PATH}/lib" ...
```

We can also set command line parameters for the python script `model.py`, which have to succeed the name of the script:

```
tarantella -- model.py --batch_size=64 --learning_rate=0.01
```

On a single node, we can also explicitly specify the number of TensorFlow instances we want to use. This is done with the `-n` option:

```
tarantella -n 2 -- model.py --batch_size=64
```

Here, `tarantella` will try to execute distributedly on 2 GPUs. If there are not enough GPUs available, `tarantella` will print a `WARNING` and run 2 instances of TensorFlow on the CPU instead.

Next, let's run `tarantella` on multiple nodes. In order to do this, we need to provide `tarantella` with a `hostfile` that contains the `hostname` s of the nodes that we want to use:

```
$ cat hostfile
name_of_node_1
name_of_node_2
```

Note: On the **STYX** cluster, the list of hostnames that belong to a job can be generated by running the following command:

```
echo ${CARME_NODES} | uniq > ./hostfile
```

Caution: Create a job comprising multiple nodes to run Tarantella distributedly! Only nodes that belong to the same job can be accessed by the `tarantella` command.

With this `hostfile` we can run `tarantella` on multiple nodes:

```
tarantella --hostfile hostfile -- model.py
```

In this case, `tarantella` uses *all* GPUs it can find. If no GPUs are available, `tarantella` will start *one* TensorFlow instance per node on the CPUs, and will issue a `WARNING` message. Again, this can be disabled by explicitly using the `--no-gpu` option.

As before, you can specify the number of GPUs/CPUs used per node explicitly with the option `--n-per-node <number>`:

```
tarantella --hostfile hostfile --n-per-node 2 --no-gpu -- model.py --batch_size=64
```

In this example, `tarantella` would execute 2 instances of TensorFlow on the CPUs of each node specified in `hostfile`.

In addition, `tarantella` can be run with different levels of logging output. The log-levels that are available are `INFO`, `WARNING`, `DEBUG` and `ERROR`, and can be set with `--log-level`:

```
tarantella --hostfile hostfile --log-level INFO -- model.py
```

To add your own environment variables, add `-x ENV_VAR_NAME=VALUE` to your `tarantella` command. This option will ensure the environment variable `ENV_VAR_NAME` is exported on all ranks before executing the code. An example is shown below:

```
tarantella --hostfile hostfile -x DATASET=/scratch/data TF_CPP_MIN_LOG_LEVEL=1 -- model.
↪py
```

Both `DATASET` and `TF_CPP_MIN_LOG_LEVEL` will be exported as environment variables before executing `model.py`, in the same order they were specified to the command line.

To terminate a running `tarantella` instance, execute another `tarantella` command that specifies the `--cleanup` option in addition to the name of the program you want to interrupt.

```
tarantella --hostfile hostfile --cleanup -- model.py
```

The above command will stop the `model.py` execution on all the nodes provided in `hostfile`. You can also enable the `--force` flag to immediately terminate unresponsive processes.

Note: Any running `tarantella` execution can be terminated by using `Ctrl+c`, regardless of whether it was started on a single node or on multiple hosts.

2.3 Using distributed datasets

This section explains how to use Tarantella's distributed datasets.

The recommended way in which to provide your dataset to Tarantella is by passing a *batched* `tf.data.Dataset` to `tnt.Model.fit`. In order to do this, create a `Dataset` and apply the `batch` transformation using the (global) batch size to it. However, do not provide a value to `batch_size` in `tnt.Model.fit`, which would lead to double batching, and thus modified shapes for the input data.

Tarantella can distribute any `tf.data.Dataset`, regardless of the number and type of transformations that have been applied to it.

Note: When using the `dataset.shuffle` transformation without a `seed`, Tarantella will use a fixed default `seed`.

This guarantees that the input data is shuffled the same way on all devices, when no `seed` is given, which is necessary for consistency. However, when a random `seed` is provided by the user, Tarantella will use that one instead.

Tarantella does not support any other way to feed data to `fit` at the moment. In particular, Numpy arrays, TensorFlow tensors and generators are not supported.

Tarantella's automatic data distribution can be switched off by passing `tnt_distribute_dataset = False` in `tnt.Model.fit`, in which case Tarantella will issue an `INFO` message. If a validation dataset is passed to `tnt.Model.fit`, it should also be batched with the global batch size. You can similarly switch off its automatic micro-batching mechanism by setting `tnt_distribute_validation_dataset = False`.

2.4 Callbacks

Tarantella callbacks are discussed in detail in the Tarantella docs.

CHAPTER

THREE

IMAGE CLASSIFICATION WITH TARANTELLA

This section delves into a more advanced usage of Tarantella by looking at distributed training for state-of-the-art image classification models.

The image classification model architectures are imported through the tf.keras.applications module, available in recent TensorFlow releases.

3.1 ResNet-50

Deep Residual Networks (ResNets) represented a breakthrough in the field of computer vision, enabling deeper and more complex deep convolutional networks. Introduced in [He], ResNet-50 has become a standard model for image classification tasks, and has been shown to scale to very large number of nodes in data parallel training [Goyal].

3.1.1 Run Resnet-50 with Tarantella

Let's assume we have access to two nodes (saved in `hostfile`) equipped with 4 GPUs each. We can now simply run the ResNet-50 as follows:

```
cd examples/image_classification
tarantella --hostfile ./hostfile --devices-per-node 4 \
-- ./train_imagenet_main.py --model_arch=resnet50 --batch_size=256 --train_epochs=3 \
                            --val_freq=3 --train_num_samples=2560 --val_num_samples=256 \
                            --synthetic_data
```

The above command will train a ResNet-50 models on the 8 devices available in parallel for 3 epochs. The `--val_freq` parameter specifies the frequency of evaluations of the *validation dataset* performed in between training epochs.

Note the `--batch_size` parameter, which specifies the global batch size used in training.

The `--synthetic_data` instructs the code to generate a synthetic ImageNet-like dataset, that can be used to showcase the training procedure. However, it will not provide any meaningful results. Remove the `--synthetic_data` parameter a specify a `--data_dir` path to an actual ImageNet directory to properly train the model.

Note: On the STYX cluster, a pre-downloaded version of the ImageNet dataset can be found in `/home/DATA/PUBLIC_DATA/ImageNet`.

Caution: On **STYX**, don't forget to add the following `-x` flags to the `tarantella` command to correctly detect the GPUs (and redo the steps from *here* if in a new terminal)

```
tarantella -x XLA_FLAGS="--xla_gpu_cuda_data_dir=${CONDA_ENV_PATH}/lib" ...
```

3.1.2 Implementation overview

We will now look closer into the implementation of the ResNet-50 training scheme. The main training steps reside in the `examples/image_classification/train_imagenet_main.py` file.

The most important step in enabling data parallelism with Tarantella is to wrap the Keras model into a Tarantella model that uses data parallelism for speeding up training.

This is summarized below for the *ResNet50* model:

```
model = tf.keras.applications.resnet50.ResNet50(include_top=True, weights=None,␣
↪classes=1000,
                                                input_shape=(224, 224, 3), input_
↪tensor=None,
                                                pooling=None, classifier_activation=
↪'softmax')
model = tnt.Model(model,
                  parallel_strategy = tnt.ParallelStrategy.DATA)
```

Next, the training procedure can simply be written down as it would be for a standard, TensorFlow-only model. No further changes are required to train the model in a distributed manner.

In particular, the ImageNet dataset is loaded and preprocessed as follows:

```
train_input_dataset = load_dataset(dataset_type='train',
                                   data_dir=args.data_dir, num_samples = args.train_num_
↪samples,
                                   batch_size=args.batch_size, dtype=tf.float32,
                                   drop_remainder=args.drop_remainder,
                                   shuffle_seed=args.shuffle_seed)
```

The load_dataset function reads the input files in `data_dir`, loads the training samples, and processes them into TensorFlow datasets.

The user only needs to pass the global `batch_size` value, and the Tarantella framework will ensure that the dataset is properly distributed among devices, such that:

- each device will process an independent set of samples
- each device will group the samples into micro batches, where the micro-batch size will be roughly equal to `batch_size / num_devices`. If the batch size is not a multiple of the number of ranks, the remainder samples will be equally distributed among the participating ranks, such that some ranks will use a micro-batch of `(batch_size / num_devices) + 1`.
- each device will apply the same set of transformations to its input samples as specified in the `load_dataset` function.

The advantage of using the *automatic dataset distribution* mechanism of Tarantella is that users can reason about their I/O pipeline without taking care of the details about how to distribute it.

Before starting the training, the model is compiled using a standard Keras optimizer and loss.

```
model.compile('optimizer' : tf.keras.optimizers.SGD(learning_rate=lr_schedule,␣
↪momentum=0.9),
              'loss' : tf.keras.losses.SparseCategoricalCrossentropy(),
              'metrics' : [tf.keras.metrics.SparseCategoricalAccuracy()])
```

We provide flags to enable the most commonly used Keras `callbacks`, such as the `TensorBoard` profiler, which can simply be passed to the `fit` function of the Tarantella model.

```
callbacks.append(tf.keras.callbacks.TensorBoard(log_dir = flags_obj.model_dir,
                                                profile_batch = 2))
```

There is no need for any further changes to proceed with distributed training:

```
history = model.fit(train_dataset,
                    validation_data = val_dataset,
                    validation_freq=args.val_freq,
                    epochs=args.train_epochs,
                    callbacks=callbacks,
                    verbose=args.verbose)
```

References

CHAPTER

FOUR

FREQUENTLY ASKED QUESTIONS (FAQ)

Check this page out for frequently asked questions about Tarantella.

BIBLIOGRAPHY

[Goyal] Goyal, Priya, et al. “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour.” arXiv preprint arXiv:1706.02677 (2017).

[He] He, Kaiming, et al. “Deep residual learning for image recognition.” Proceedings of the IEEE conference on computer vision and pattern recognition. arXiv preprint arXiv:1512.03385 (2016).