Deep Learning on HPC Workshop 2021

Oliver Cairncross

Research Computing Centre

Preamble

The Deep Learning on HPC workshop has been designed for experienced users that require, or would benefit from, using high performance computing resources to carry out their DL work. Wiener, named after the mathematician Norbert Wiener, is a high performance computer purchased by UQ especially to support large scale image processing and machine learning. It has the following attributes which makes it a great platform for large DL tasks:

- accelerators for high speed numerical computation (GPU's);
- fast, high capacity file systems;
- ample memory;
- high speed networking; and
- a job queuing system.

The workshop is based on a case study used to demonstrate various concepts required to convert single process DL training code into multi process (parallel GPU) code. Given it isn't feasible to address even a fraction of any real world examples; the workshop is restricted to a single DL framework—Keras; as well as a single use case—image classification. Nevertheless, participants should gain an understanding of how to implement parallel GPU DL code on HPC systems.

It is expected that participants already have experience with:

- writing and running Python based deep learning code;
- python environment management; and
- the Linux command line interface.

Although this workshop is tailored to a specific HPC cluster the code can be used on any multi GPU system.

Different versions of this material will be made available for other platforms such as Massive found at Monash University in due course.

Topics Covered

- **The Deep Learning Environment**
  Anaconda will be used to manage the environment. Special consideration is required as to where files and data are located given Wiener's file space constraints.
- **Using Multiple GPUs**
  - Multiple GPUs can be used in parallel to speed up the time required to train deep learning models. Horovod is a framework we will use to undertake this task as it simplifies the task of modifying existing code for parallel execution.
  - The speed gains may not be always realised due to the nature of models and there can even be undesirable side effects with respect to convergence.
  - It is also possible to use multiple GPUs to train models that are too large for a single GPU but this will not be covered. This scenario will be addressed sometime in the future given support for large models is becoming important.
- **Job Queues**
  Jobs are run by submitting them to the Slurm queuing system. Slurm allows many jobs to be submitted simultaneously; perhaps with different hyper parameters and/or data.
- **Monitoring Jobs**
  Running jobs may need monitoring. Familiar tools, such as Tensorboard, can be used for this but specific steps are required to run such tools on HPC.
- **Analysing Performance**
  Performing basic analysis of how efficiently a deep learning job has run and checking if the environment is configured correctly.

HPC Deep Learning Environment

Setup Anaconda

Anaconda is the package manager used to create and manage your DL Python environments. On Wiener the latest version of Anaconda is 4.10.3 supporting Python 3.7.6.

If you haven't already configured Anaconda for you account on Wiener run these commands on your command prompt:

```
$ module load anaconda/3.7
$ conda init
```

Anaconda has added the conda command to your shell. Your shell's startup file (.bashrc if you're using the default shell) has been modified to support this by adding lines enclosed by

```
# >>> conda initialize >>>; and
# <<< conda initialize <<<
```

Test that Anaconda has been configure by running `conda info`. If Anaconda is configured properly the attributes of your environment will be printed. The first two attributes will be similar to:

```
active environment : base
active env location : /opt/ohpc/pub/apps/...
```

Create a Deep Learning Environment

The next step is to create an Anaconda environment to support DL code. These environments require a lot of disk space and shouldn't be placed in home directories given the 5GB quota imposed on `/home`. It's recommended that you store environments on Wiener's `/scratch` volume and configure anaconda to use this location by default.

Create a directory on the scratch volume to store the environment with the following (recommended) structure:

```
/scratch/ou_designation/username/anaconda
```

scratch: large storage volume for projects.

ou_designation: your organisational unit such as rcc, cai, imb, itee, etc.

username: your user name (which may not exist).

anaconda: put all your anaconda data here.

HPC Deep Learning Environment

Configure anaconda to use your new storage location by creating a `~/.condarc` if it doesn't exist and adding the following lines to it:

`auto_update_conda: False`	We don't want anaconda to update itself as it is read only

`pkgs_dirs:`
`  - /scratch/ou_designation/username/anaconda/pkgs`
 	
`envs_dirs:`
`  - /scratch/ou_designation/username/anaconda/envs`
	The package cache and environments are placed in our scratch directory by default

The module system sets an environment variable, CONDA_ENVS_PATH, which overrides the file locations specified in your `~/.condarc` file. Unset this variable by executing `unset CONDA_ENVS_PATH` on the comment line. Unsetting this variable is required every time the `module load anaconda` command is run. Fortunately, because of the modifications made to your startup script, you no longer need use module system to load anaconda—it will be available by default.

HPC Deep Learning Environment

Finally we create the horovod environment used to run out multi-GPG code.

```
$ conda create --clone /opt/ohpc/pub/apps/horovod_2021/horovod-gpu-data-science-project/env --name data-science
```

Note that we are cloning an existing environment. This ensures that everything is configured properly to run on Wiener. Creating the environment will take some time.

When finished activate the environment and we are ready to train models using multiple GPUs.

```
$ conda activate data-science
```

Deep Learning with Multiple GPUs

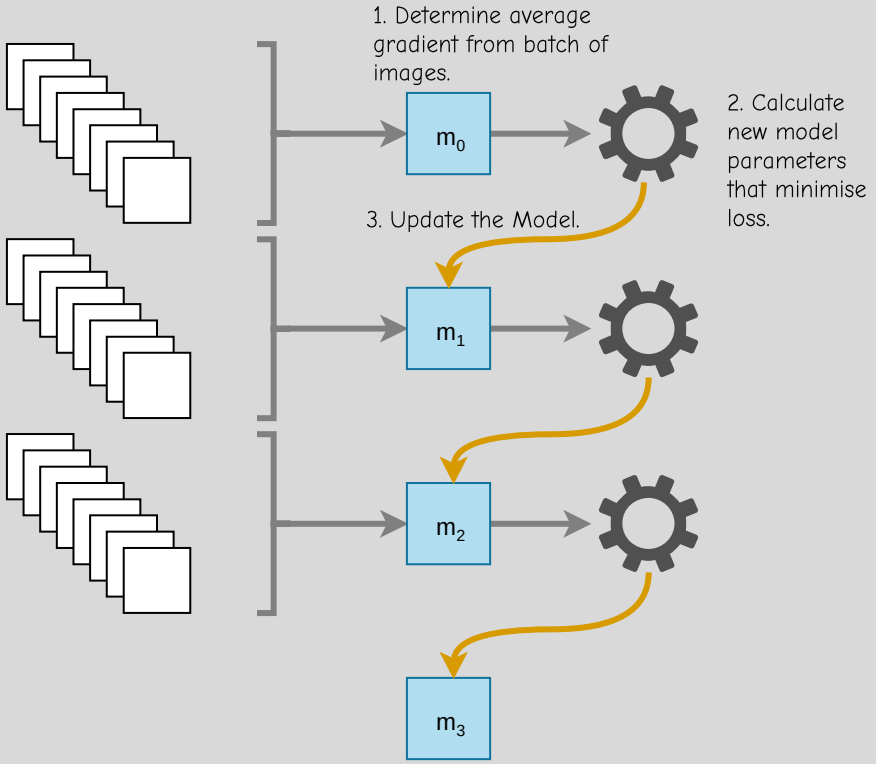
Traditionally, implementing parallel GPU training is done using a parameter server architecture. The parameter server's role is to distribute data, calculate average gradients, update the model and coordinate the individual GPU process. This architecture requires that existing code is modified to incorporate the role of the parameter server.

The Horovod framework uses a different architecture known as ring-all-reduce which does not require a parameter server. Instead, the GPU processes communicate with each other directly in a circular fashion. The tasks of averaging gradients and model updates are shared by all processes in a ring (roughly speaking). Removing the parameter server makes it easier to modify existing code to use multiple GPUs. Incidentally, the framework is called Horovod for its likeness to the Slavic dance of the same name.

SGD Overview

To motivate the discussion we will consider the Stochastic Gradient Descent process applied to image classification. First we will consider the simple single process approach and move on to a parallel one.

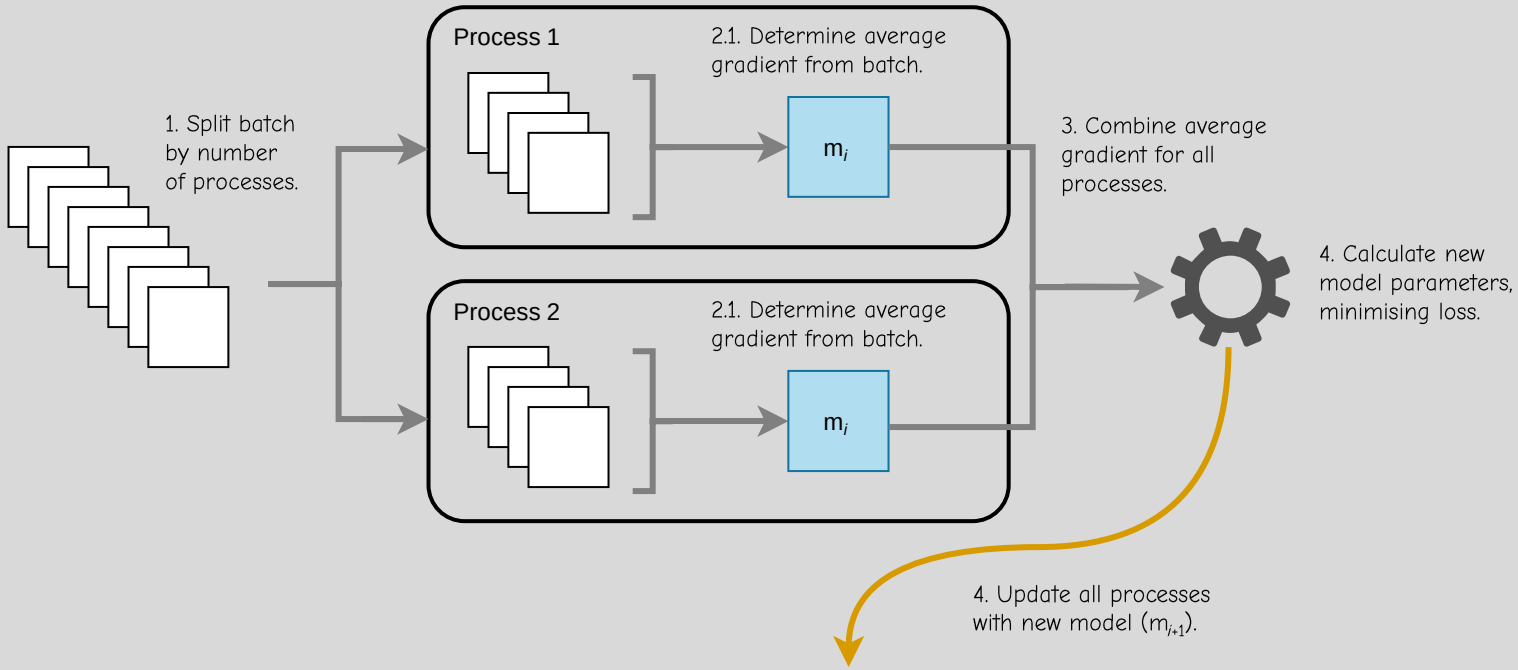
- Image frames are grouped into batches
- Average loss is calculated for a batch
- Model updated in order to minimise loss
- Repeat until all frames are processed (one epoch).



Parallel SGD Logical Overview

Parallel SGD is implemented by subdividing the work and allocating it to multiple processes that run concurrently. The following diagram depicts two processes computing the average gradient for a subset of a given batch that are later combined to determine the average gradient for the entire batch. Given the processes work in parallel, the batches are processed more quickly.

1. Each process:
   a) reads a mini-batch
   b) passes it through the model
   c) compute the mini-batch gradient
2. Average gradient is computed for all mini-batch gradients
3. Model is updated (identical for all processes
4. Repeat the process (go to step 1)



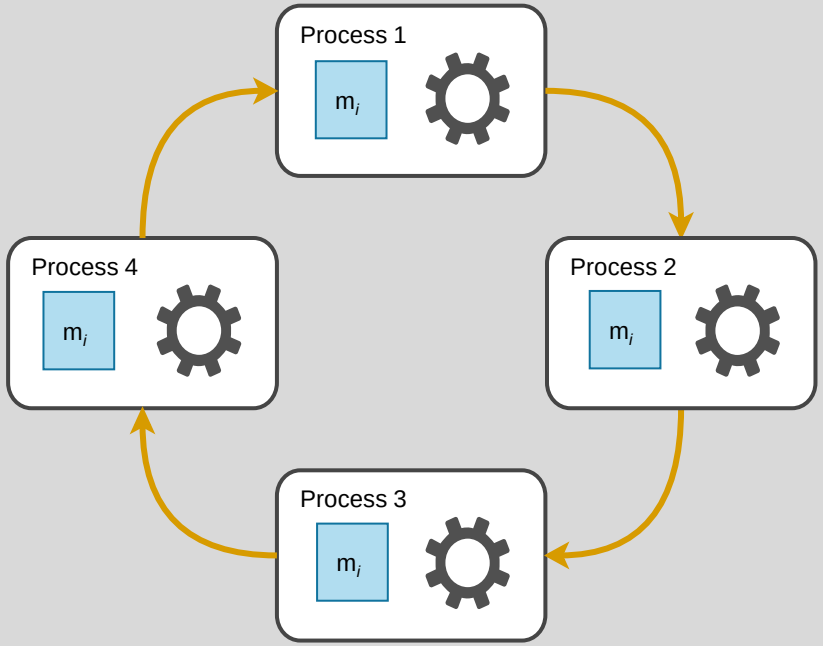
Parallel SGD Implementation*

* Horovod's approach to parallel processing is discussed here. There are other popular approaches, such as parameter servers, which differ significantly to the approach used by Horovod.

The Horovod framework provides mechanisms that create and manage the processes used to implement parallel deep learning. These processes have the following characteristics:

- Independent, self-contained code that run on hardware.
- Compute the gradient on subsets of images independently.
- Work together to calculate average gradient and new model parameters.
- Contain the same copy of the model at all times.
- Communicate with each other in uniform, ordered manner.
- One process is allocated to one GPU.

The key point to remember is that Horovod automates the bulk of the work needed for this for us.



Horovod Code

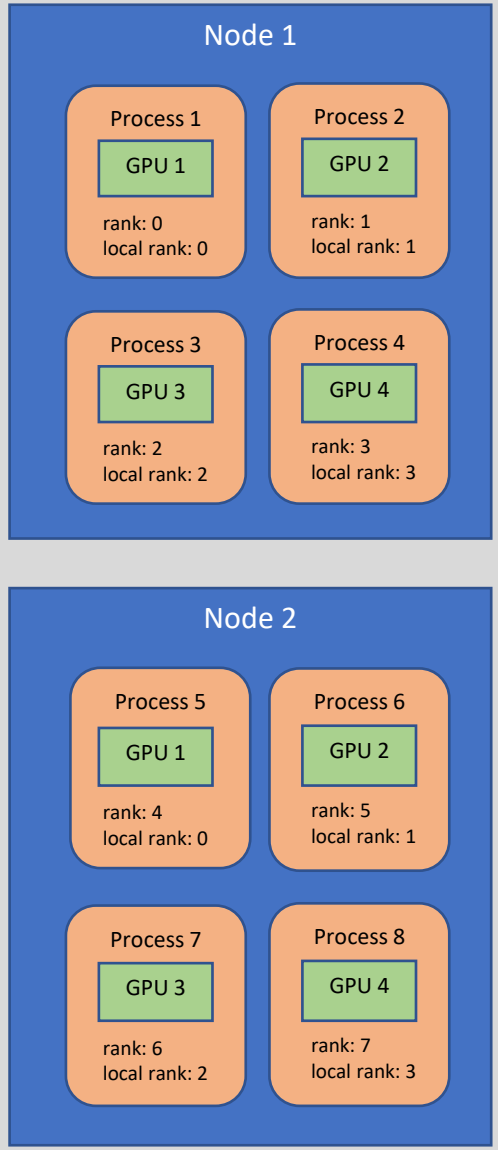
Implementing multi GPU deep learning is surprisingly straight forward with the Horovod framework. The important points to keep in mind are:

- Multiple, identical, copies of the code will run. The running code is referred to as a process and one process is run per GPU.
- Horovod provides information so that processes can identify where they are with respect to each other and where they are running on HPC clusters.

  Rank: Sequential number starting at 0 across all processes. Used typically to identify the first process.

  Local Rank: Sequential number for the processes in a node. Used to tie a process to a unique GPU.

  Size: The total number of processes that are running.
- Each process will work on a subset of that data that makes up a batch
- Allocating data to the processes; calculating the average gradient; and updating the model is done automatically by Horovod. We don't need to worry about doing this.
- We need to take special care with tasks that generate output; such as checkpoints and Tensorboard logs.



Horovod Code

To understand the Horovod framework we will modify code that trains a simple image classification model using stochastic gradient descent on a single GPU. The code consist of a utility module and an executable script.

utility
• load the dataset • transform the dataset so that it can be used in the CNN network • functions to define output locations • define the layers of the CNN model

fashion-minst
• calls utitility.py to setup the environment • defines a callback for checkpointing • compiles the model • trains the models and saves it

The parallel implementation is provided by mutli-fashion-minst. We are using the fashion-minst dataset for this workshop which is relatively small and this allows us to train models very quickly. It would not be worthwhile to implement a parallel deep learning on such a dataset in the real world.

Running Horovod Code

It is worthwhile to describe how the parallel processes are launched on HPC clusters before discussing the code itself. This will help us understand how to configure our processes and how the processes get information such as rank, local rank, etc.

Wiener uses the Slurm workload manager to launch and manage jobs. A batch script with special parameters tells Slurm the configuration we want to use. This script is submitted using the sbatch command.

A sample script job.sh has been provided with our code samples. This script tells Slurm that we want to run a job using four GPUs.

#!/bin/bash	Specify that this is a bash script.
#SBATCH --job-name=SGD-Train	A job a name is useful to identify to jobs in the queuing system.
#SBATCH --output=slurm-%j.out **#SBATCH --error=slurm-%j.err**	Errors and output for running processes will be sent to these files. By default Slurm will write errors and errors to the same file; here two files will be used. These files are created in the directory that the sbatch command was run. If you want to place them elsewhere the directories must already exist. The job id will be appended to the filenames.
#SBATCH --nodes=1	Number of nodes to use combined with the gres option (bellow) will determine the number of processes allocated to run the job.
#SBATCH --gres=gpu:tesla-smx2:4	Determines what type of node will be allocated. Wiener has two types of node for running deep learning jobs: **tesla**: a maximum of 2 GPUs per node, 16GB of GPU memory per GPU **tesla-smx2**: a maximum of 4 GPUs per node, 32GB of GPU memory per GPU The syntax is gres=gpu:*node_type*:*number_of_gpus.* The number of GPUs is per node. Multiply the number of nodes requested by the number of GPUs requested for total number of GPUs allocated.
#SBATCH --ntasks-per-node=4	The number of tasks per node. This needs to match the number of GPUs per node. Number of tasks is the same as the number of processes that will run.
#SBATCH --cpus-per-task=7	All nodes on Wiener have 28 CPUs. Setting 7 CPUs per GPU is a simple way to scale CPUs with GPU usage. If all GPUs in a node are *to b*e used we will use all the available CPUs as well.
#SBATCH --mem-per-cpu=12G	Memory is scaled using the same principal as CPUs. 345G is the maximum practical limit when using four GPUs.
#SBATCH --partition=LMA	The tesla and tesla-smx2 nodes used for the workshop are in the LMA partition of the cluster.

Running Horovod Code

The latter section of the script is concerned with setting up the python and Horovod environment used by all the processes. The last part launches the processes with the mpiexec command.

Mpiexec starts the processes and ensures that environment variables are passed to them. It also passes runtime information to Horovod such as rank and local rank and total number of processes.

The script is submitted to the queueing system by executing sbatch run-job.sh from the command line.

module load anaconda/3.7 **module load cuda/11.3.0**	Load the required modules required to run our code
unset CONDA_ENVS_PATH	**Important!** Wiener's module system sets this environment variable. It must be cleared in order to for our .condarc file to be used properly.
export HOROVOD_CUDA_HOME=$CUDA_HOME **export HOROVOD_GPU_OPERATIONS=NCCL** **export NCCL_DEBUG=WARN**	Create and set Horovod specific environment variables.
eval "$(conda shell.bash hook)" **conda activate data-science** **cd /scratch/rcc/$USERNAME/project/ml-hpc**	Active the conda environment that supports Horovod and set the current directory to where the training scripts are located.
mpiexec -np ${SLURM_NTASKS} ** **-bind-to none -map-by slot ** **python multi-fashion-minst.py	Execute the code using multiple processes. The bind slot option enables GPU to process mappings

Horovod Code

These are the changes made to the single process code in order to train the model with multiple GPUs. This section is concerned with initialising the process and binding one GPU to each process.

`import horovod.tensorflow.keras as hvd`	Import the library. Here we import the Keras version.
`hvd.init()`	Initialise Horovod.
`gpus = tf.config.experimental.list_physical_devices('GPU')`	Obtain a list of GPUs on the node that this process is running on.
`for gpu in gpus:` `  tf.config.experimental.set_memory_growth(gpu, True)`	Do not use all the GPU memory at once. All process set this for all GPUs on a node even though they will only use one GPU. This is because TensorFlow will reserve all memory on all GPUs for any process by default.
`if gpus:` `  tf.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU')`	Bind this process identified by local rank to a single GPU. The Slurm script that runs the code ensures that there is a one to one relationship between the local rank of the process and GPUs on the node.

Horovod Code

Horovod does the work such batching, distribution of data, and calculation of average gradients via the optimiser. The compile_model function is modified to facilitate this.

`opt = hvd.DistributedOptimizer(SGD(lr=learning_rate, momentum=momentum))`	The optimiser is where Horovod distributes image frames to processes and later recombines them. One line of code wraps a standard Keras optimiser in an Horovod optimiser to accomplish

`model.compile(optimizer=opt,`
`  loss='categorical_crossentropy',`
`  metrics=['accuracy'],`
`  experimental_run_tf_function=False)`
	The experimental_run_tf_function parameter is set to false to ensure the Horovod optimiser is used and not the underlying Karas optimiser.

Horovod Code

Callbacks are the mechanism used by Keras to coordinate the training process. Essentially they are list of functions that are called at the end of every epoch.

`if hvd.rank() == 0:`
`  cb.append(tf.keras.callbacks.ModelCheckpoint(checkpoint_filename))`
`  cb.append(tf.keras.callbacks.TensorBoard(log_dir=tensorboard_log_dir,`
`  histogram_freq=1))`
	Ensuring that only one process generates output. It is important to remember that the model is always identical on all processes. As output only pertains to the model only one process is required to generate the output.

`cb.append(hvd.callbacks.BroadcastGlobalVariablesCallback(0))`
`cb.append(hvd.callbacks.MetricAverageCallback())`
`cb.append(`*`hvd.callbacks.LearningRateWarmupCallback…`*`)`
	The Broadcast callback ensures the same initial model is used when training starts.
	The Metric average callback ensures the model is synchronised before Tensorboard data is generated.
	The Learning Rate Warmup callback is discussed later.

`if hvd.rank() == 0:` `  model.save(model_filename)`	Finally the model is saved. Not in a callback, but when the fit process completes.

Horovod Code

Changes are required to model training hyperparameters in order to make it run efficiently on multiple GPUs. If a batch size of 32 is split into 4 minibatches (i.e., 8 images) the processes can end up spending more time communicating with each other to coordinate the task then doing actual computations. This can negate the speed improvements that a multi process implementation is supposed to achieve. In some cases the multi process implementation can take even longer.

Making such changes to hyperparameters will impact model convergence and sometimes in negative ways. It is for the machine learning specialist to determine what is feasible depending on factors such as model composition; optimisation method; source data etc.

In our case we have tuned the parameters based on research done on our type of data and optimiser. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

Experimentation may be required to determine what changes need be made so that the job runs efficiently in a multi GPU environment. In some cases it may not be possible.

batch_size *= hvd.size() learning_rate *= hvd.size()	The batch and learning rates are scaled by the number of processes (i.e., GPUs) used. This provides a larger batch size and results in less communication overhead.
cb.append(hvd.callbacks.LearningRateWarmupCallback(initial_lr=learning_rate / hvd.size(), warmup_epochs=3, verbose=1))	From the previous code block The learning rate is scaled up over the span of three epochs

Monitoring the Job

The training process can be monitored via Tensorboard. Our code is configured to generate files for this purpose in the out/timestamp/tb_log directory.

Monitor a running job

On Wiener start a command prompt and activate the anaconda data-science environment.

To start the Tensorboard server execute the following command:

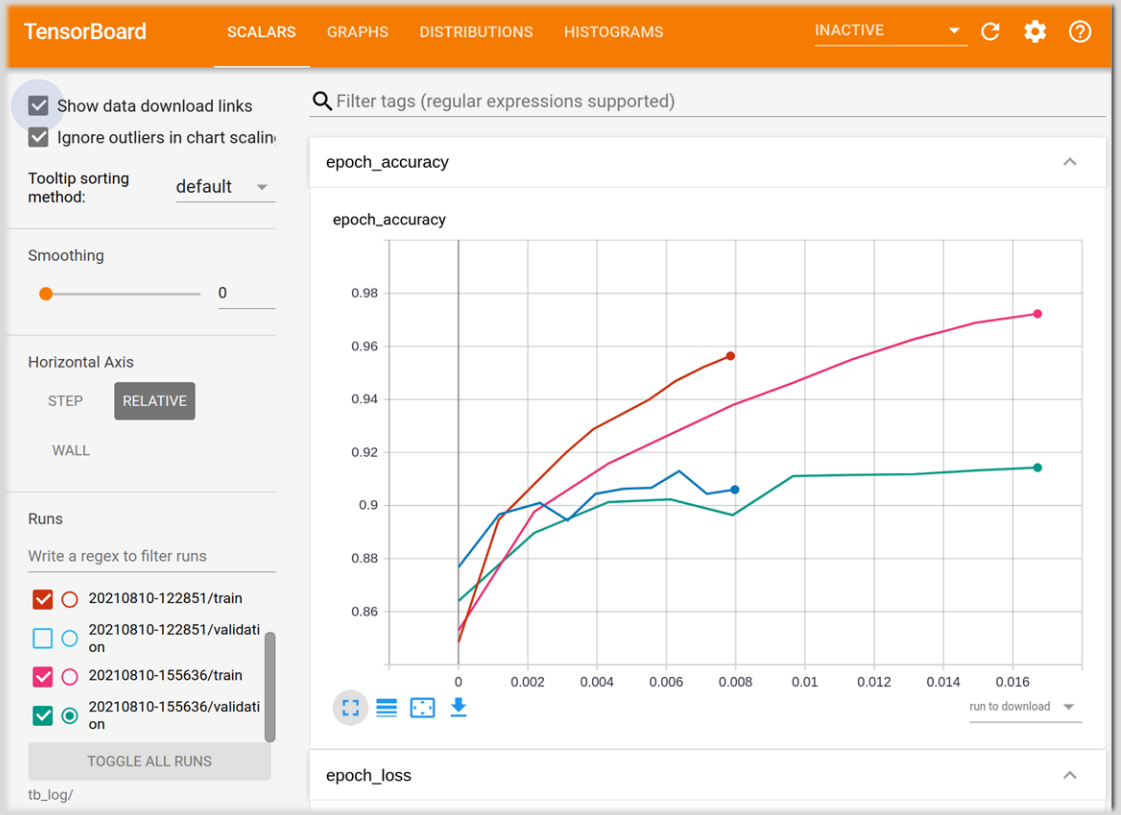
```
$ tensorboard --logdir path_to_your/tb_log --bind_all
```

On your workstation create a path to your Tensorboard server execute the following comment (Linux and Mac):

```
ssh L:6006:localhost usernme@wiener.hpc.dc.uq.edu.au
```

Open you browser and go to http://localhost:6006

Tensorboard will open and you can monitor the running job. You will be able to see updates as they happen.



The graph plots the training and validation accuracy of two and four GPU job. The four GPU job is twice as fast with similar accuracy.

Analysing Performance

Horovod has the ability to create profile logs that can be used to give an idea as to how efficiently resources are being utilised.

To create a log we add the following lines to the mpiexec command of our sbatch script.

```
mpiexec -np ${SLURM_NTASKS} \
    -env HOROVOD_TIMELINE=timeline.json \
    -env HOROVOD_TIMELINE_MARK_CYCLES=0 \
    -bind-to none -map-by slot \
    python multi-fashion-minst.py
```

When the job is run the first process will output a file called timeline.json which needs to be copied to your workstation.

Open the file which a Chrome browser (it must be Chrome) by entering chrome://tracing in the address bar.

A trace diagram will be displayed. The diagram can be used to quickly eyeball how the job has run. For example we can:

- see how much time is spent negotiating communications vs computation.
- check that features such that NCCL are used.

We won't cover performance analysis deeply in this workshop as it would require a great deal of time.