ChainerMN Documentation

Release 1.0.0b1

Takuya Akiba

Aug 31, 2017

Contents

1 Installation **3**
- 1.1 Installation Guide . . . 3
- 1.2 Step-by-Step Troubleshooting . . . 5

2 Tutorial **13**
- 2.1 Overview . . . 13
- 2.2 Step 1: Communicators and Optimizers . . . 14
- 2.3 Step 2: Datasets and Evaluators . . . 16
- 2.4 Tips and FAQs . . . 18

3 API Reference **19**
- 3.1 Communicators . . . 19
- 3.2 Optimizers and Evaluators . . . 19
- 3.3 Dataset Utilities . . . 20

Python Module Index **21**

Contents:

CHAPTER 1

Installation

Installation Guide

Requirements

In addition to Chainer, ChainerMN depends on the following software libraries: CUDA-Aware MPI, NVIDIA NCCL, and a few Python packages including MPI4py.

Chainer

ChainerMN adds distributed training features to Chainer; thus it naturally requires Chainer. Please refer to the official instructions to install.

CUDA-Aware MPI

ChainerMN relies on MPI. In particular, for efficient communication between GPUs, it uses CUDA-aware MPI. For details about CUDA-aware MPI, see this introduction article. (If you use only the CPU mode, MPI does not need to be CUDA-Aware. See *Non-GPU environments* for more details.)

The CUDA-aware features depend on several MPI packages, which need to be configured and built properly. The following are examples of Open MPI and MVAPICH.

Open MPI (for details, see the official instructions):

```
$ ./configure --with-cuda
$ make -j4
$ sudo make install
```

MVAPICH (for details, see the official instructions):

```
$ ./configure --enable-cuda
$ make -j4
$ sudo make install
$ export MV2_USE_CUDA=1  # Should be set all the time when using ChainerMN
```

NVIDIA NCCL

To enable efficient intra-node GPU-to-GPU communication, we use NVIDIA NCCL. See the official instructions for installation.

Please properly configure environment variables to expose NCCL both when you install and use ChainerMN. Typical configurations should look like the following:

```
export NCCL_ROOT=<path to NCCL directory>
export CPATH=$NCCL_ROOT/include:$CPATH
export LD_LIBRARY_PATH=$NCCL_ROOT/lib/:$LD_LIBRARY_PATH
export LIBRARY_PATH=$NCCL_ROOT/lib/:$LIBRARY_PATH
```

ChainerMN requires NCCL even if you have only one GPU per node. The only exception is when you run ChainerMN on CPU-only environments. See *Non-GPU environments* for more details.

MPI4py

ChainerMN depends on a few Python packages, which are automatically installed when you install ChainerMN.

However, among them, we need to be a little careful about MPI4py. It links to MPI at installation time, so please be sure to properly configure environment variables so that MPI is available at installation time. In particular, if you have multiple MPI implementations in your environment, please expose the implementation that you want to use both when you install and use ChainerMN.

In addition, Cython may not be installed automatically. It can be installed manually via **`pip`**:

```
$ pip install cython
```

Installation

Install via pip

We recommend to install ChainerMN via **`pip`**:

```
$ pip install chainermn
```

Install from Source

You can use `setup.py` to install ChainerMN from source:

```
$ tar zxf chainermn-x.y.z.tar.gz
$ cd chainermn-x.y.z
$ python setup.py install
```

Non-GPU environments

For users who want to try ChainerMN in a CPU-only environment, typically for testing for debugging purpose, ChainerMN can be built with the `--no-nccl` flag.:

```
$ python setup.py install --no-nccl
```

In this case, the MPI does not have to be CUDA-aware. Only `naive` communicator works with the CPU mode.

Step-by-Step Troubleshooting

This section is a step-by-step troubleshooting guide for ChainerMN. Please follow these steps to identify and fix your problem.

We assume that you are using Linux or another Unix-like environment.

Single-node environment

Basic MPI installation

Although ChainerMN stands for "Chainer MultiNode," it is good to start from single-node execution. First of all, you need MPI. If MPI is correctly installed, you will see the **`mpicc`** and **`mpiexec`** commands in your PATH.

Below is an example of the output from Mvapich on Linux.:

```
$ which mpicc
/usr/local/bin/mpicc

$ mpicc -show
gcc -I/usr/local/include ...(snip)... -lmpi

$ which mpiexec
/usr/local/bin/mpiexec

$ mpiexec --version
HYDRA build details:
Version:                                 3.1.4
Release Date:                            Wed Sep  7 14:33:43 EDT 2016
CC:                              gcc
CXX:                             g++
F77:
F90:
Configure options:  (snip)
Process Manager:                         pmi
Launchers available:                     ssh rsh fork slurm ll lsf sge manual persist
Topology libraries available:            hwloc
Resource management kernels available:   user slurm ll lsf sge pbs cobalt
Checkpointing libraries available:
Demux engines available:                 poll select
```

If you see any error in above commands, please go back to the *CUDA-Aware MPI* and check your MPI installation.

Check what MPI you are using

In *CUDA-Aware MPI*, we mention both of *Open MPI* and *Mvapich*. If the MPI is provided by the system administrator and you are not really sure which MPI you are using, check the output of *mpiexec –version*.

- If the output contains *HYDRA*, then it's MVAPICH (or possibly MPICH).
- If the output contains *OpenRTE*, then it's Open MPI.

However, in such a case, you should make sure that the MPI is *CUDA-aware*, as mentioned below. We recommend to build your own MPI.

Check if MPI is CUDA-aware

Your MPI must be configured as *CUDA-aware*. You can use the following C program to check it.

```
/* check_cuda_aware.c */
#include <assert.h>
#include <stdio.h>
#include <mpi.h>
#include <cuda_runtime.h>

#define CUDA_CALL(expr) do {                    \
  cudaError_t err;                              \
  err = expr;                                   \
  assert(err == cudaSuccess);                   \
} while(0)

int main(int argc, char **argv) {
  int rank, size;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  int *sendbuf_d = NULL;
  int *recvbuf_d = NULL;

  CUDA_CALL(cudaMalloc((void**)&sendbuf_d, sizeof(int)));
  CUDA_CALL(cudaMalloc((void**)&recvbuf_d, sizeof(int)));
  CUDA_CALL(cudaMemcpy(sendbuf_d, &rank, sizeof(int), cudaMemcpyDefault));

  MPI_Reduce(sendbuf_d, recvbuf_d, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

  if (rank == 0) {
    int sum = -1;
    CUDA_CALL(cudaMemcpy(&sum, recvbuf_d, sizeof(int), cudaMemcpyDefault));
    if (sum == (size-1) * size / 2) {
      printf("OK.\n");
    } else {
      printf("Error.\n");
    }
  }

  cudaFree(sendbuf_d);
  cudaFree(recvbuf_d);
```

```
  MPI_Finalize();
}
```

Save the code to a file named `check_cuda_aware.c`. You can compile and run it with the following command.:

```
$ export MPICH_CC=nvcc  # if you use Mvapich
$ export OMPI_CC=nvcc   # if you use Open MPI
$ $(mpicc -show check_cuda_aware.c -arch sm_53 | sed -e 's/-Wl,/-Xlinker /g' | sed -e
↪'s/-pthread/-Xcompiler -pthread/')
$ ./a.out
OK.
```

If the proglam prints *OK.*, your MPI is correctly configured.

Check mpi4py

Next, let's check that mpi4py is correctly installed. You can use the following script to check it:

```
# coding: utf-8
import os
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

for i in range(size):
  if i == rank:
    print("{} {}".format(os.uname()[1], i))
  comm.Barrier()
```

Save the script into a file named `check_mpi4py.py` and run it. The output from the script should look like this.:

```
$ mpiexec -np 4 python check_mpi4py.py
host00 0
host00 1
host00 2
host00 3
```

The script prints hostnames and ranks (process id in MPI) from each MPI process in a sequential manner. *host00* is the host name of the machine your are running the process. If you get an output like below, it indicates something is wrong with your installation.:

```
# Wrong output !
$ mpiexec -n 4 python check_mpi4py.py
host00 0
host00 0
host00 0
host00 0
```

A common problem is that the **`mpicc`** used to build `mpi4py` and **`mpiexec`** used to run the script are from different MPI installations.

Finally, run **`nosetests`** to check the single-node configuration is ready.:

```
$ nosetests
......S.S...S.S...S.S...S.S.........SS
```

```
----------------------------------------------------------------------
Ran 38 tests in 63.083s

OK (SKIP=10)
```

Multi-node environmnet

Check SSH connection and enviornment variables

To use ChainerMN on multiple hosts, you need to connect to computing hosts, including the one you are currently logged into, via ssh without password authentication (and preferably without username).:

```
$ ssh host00 'hostname'
host00   # without hitting the password

$ ssh host01 'hostname'
host01   # without hitting the password

...
```

You may get a message like this:

```
The authenticity of host 'host01 (xxx.xxx.xxx.xxx)' can't be established.
ECDSA key fingerprint is SHA256:haGUMcCeC5A8lGh1lpjpwL5dF4xCglZArhhxxxxxxxxx.
Are you sure you want to continue connecting (yes/no)?
```

This message appears when you log in a host for the first time. Just type *yes* and the message won't appear again. You need to repeat this process on all computing hosts.

Also, you need to pay attention to the environment variables on remote hosts. The MPI runtime connects to the remote hosts in *non-interactive* mode, and environment variables may differ from your interactive login sessions.:

```
$ ssh host00 'env' | grep LD_LIBRARY_PATH
# Check the values and compare it to the local value.

$ ssh host01 'env' | grep LD_LIBRARY_PATH
# Check the values and compare it to the local value.

...
```

In particular, check the following variables, which are critical to executing MPI programs:

- `PATH`
- `LD_LIBRARY_PATH`
- `MV2_USE_CUDA` (if you use MVAPICH)
- `MV2_SMP_USE_CMA` (if you use MVAPICH)

Besides, you need to make sure the same **`mpiexec`** binary is used to run MPI programs.:

```
$ ssh host00 'which mpiexec'
/usr/local/bin/mpiexec

$ ssh host01 'which mpiexec'
/usr/local/bin/mpiexec
```

All the commands should give the same **mpiexec** binary path.

Program files and data

When you run MPI programs, all hosts must have the same Python binary and script files in the same path. First, check that the python binary and version are identical among hosts. Be careful if you are using *pyenv* or *Anaconda*.:

```
$ ssh host00 'which python; python --version'
/home/username/.pyenv/shims/python
Python 3.6.0 :: Anaconda 4.3.1 (64-bit)

$ ssh host01 'which python'
/home/username/.pyenv/shims/python
Python 3.6.0 :: Anaconda 4.3.1 (64-bit)

...
```

Also, the script file (and possibly data files) must be in the same path on each host.

```
$ ls yourscript.py  # in the current directory
yourscript.py

$ ssh host00 "ls $PWD/yourscript.py"
/home/username/your/dir/yourscript.py

$ ssh host01 "ls $PWD/yourscript.py"
/home/username/your/dir/yourscript.py

...
```

If you are using NFS, everything should be okay. If not, you need to transfer all the necessary files manually.

In particular, when you run the ImageNet example in ChainerMN repository, all data files must be available on all computing hosts.

hostfile

The next step is to create a hostfile. A hostfile is a list of hosts on which MPI processes run.:

```
$ vi hostfile
$ cat hostfile
host00
host01
host02
host03
```

Then, you can run your MPI program using the hostfile. To check if the MPI processes run over multiple hosts, save the following script to a file and run it via **mpiexec**:

```
# print_rank.py
import os

from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
```

```
rank = comm.Get_rank()

for i in range(size):
  if i == rank:
    print("{} {}".format(os.uname()[1], i))
  comm.Barrier()
```

If you get an output like below, it is working correctly.:

```
$ mpiexec -n 4 --hostfile hostfile python print_rank.py
host00 0
host01 1
host02 2
host03 3
```

If you have multiple GPUs, you may want to run multiple processes on each host. You can modify hostfile and specify the number of processes to run on each host.:

```
# If you are using Mvapich:
$ cat hostfile
host00:4
host01:4
host02:4
host03:4

# If you are using Open MPI
$ cat hostfile
host00 cpu=4
host01 cpu=4
host02 cpu=4
host03 cpu=4
```

With this hostfile, try running mpiexec again.:

```
$ mpiexec -n 8 --hostfile hostfile python print_rank.py
host00 0
host00 1
host00 2
host00 3
host01 4
host01 5
host01 6
host01 7
```

You will find that the first 4 processes run on host00 and the latter 4 on host01.

You can also specify computing hosts and resource mapping/binding using command line options of mpiexec. Please refer to the MPI manual for the more advanced use of mpiexec command.

If you get runtime error:

If you get the following error messages, please check the specified section of the troubleshooting or installation guide.

```
[hostxxx:mpi_rank_0][MPIDI_CH3I_SMP_init] CMA is not available. Set MV2_SMP_USE_CMA=0␣
↪to disable CMA.
[cli_0]: aborting job:
Fatal error in PMPI_Init_thread:
```

```
Other MPI error, error stack:
MPIR_Init_thread(514)....:
MPID_Init(365)...........: channel initialization failed
MPIDI_CH3_Init(404)......:
MPIDI_CH3I_SMP_Init(2132): process_vm_readv: Operation not permitted


===================================================================================
=   BAD TERMINATION OF ONE OF YOUR APPLICATION PROCESSES
=   PID 20327 RUNNING AT hostxxx
=   EXIT CODE: 1
=   CLEANING UP REMAINING PROCESSES
=   YOU CAN IGNORE THE BELOW CLEANUP MESSAGES
===================================================================================
```

-> Check the value of `MV2_SMP_USE_CMA` (see *CUDA-Aware MPI* and *Check SSH connection and enviornment variables*).

```
[hostxx:mpi_rank_0][error_sighandler] Caught error: Segmentation fault (signal 11)

===================================================================================
=   BAD TERMINATION OF ONE OF YOUR APPLICATION PROCESSES
=   PID 20643 RUNNING AT hostxx
=   EXIT CODE: 11
=   CLEANING UP REMAINING PROCESSES
=   YOU CAN IGNORE THE BELOW CLEANUP MESSAGES
===================================================================================
YOUR APPLICATION TERMINATED WITH THE EXIT STRING: Segmentation fault (signal 11)
This typically refers to a problem with your application.
Please see the FAQ page for debugging suggestions
```

-> Check the value of `MV2_USE_CUDA` (see *CUDA-Aware MPI* and *Check SSH connection and enviornment variables*)

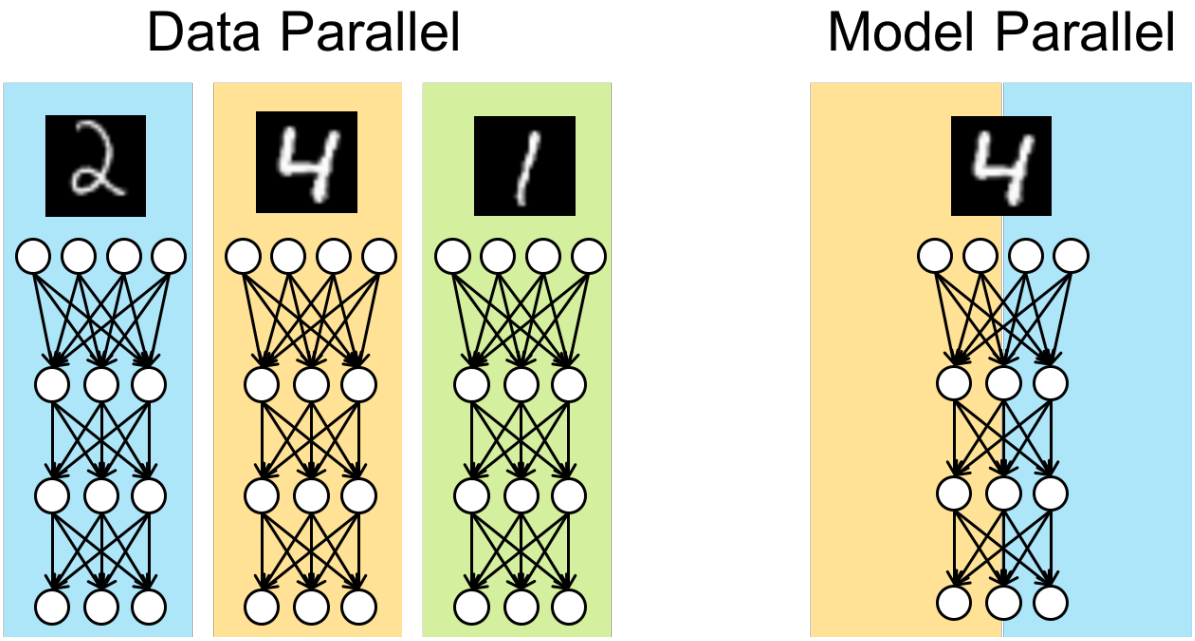
CHAPTER 2

Tutorial

Overview

Data Parallelism

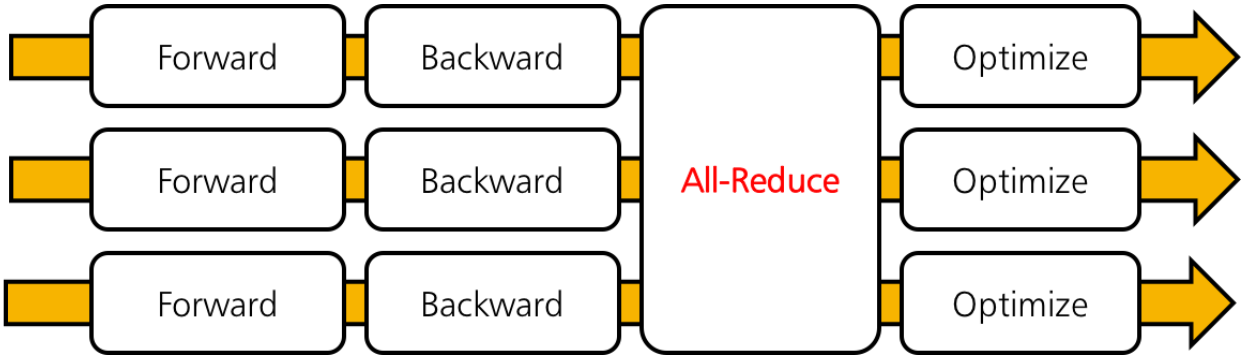
ChainerMN employs the data parallel approach for distributed training. In the data parallel approach, each worker has a model copy, and computes a gradient against a batch. Then, the workers collaborate to update the model using the gradients of all workers.



Training Iterations

What ChainerMN does for distributed training is actually quite simple. Let us look at what we do in each iteration. The following figure illustrates an iteration of standard training using Chainer (without ChainerMN). It consists of three steps: forward, backward and optimize.

When using ChainerMN, an additional step all-reduce is inserted after the backward step. In this step, workers communicate to obtain the averaged gradient over gradients of all workers. Then, the aggregated gradient is used to improve the model in the optimization step.



MPI

ChainerMN is built on MPI. MPI invokes our training script in the SPMD (single program, multiple data) way. ChainerMN is designed to create a process on each GPU. For example, let us suppose you have two nodes with four GPUs each, and want to run `train_imagenet.py`. Then, you will invoke eight Python processes running `train_imagenet.py` by using `mpiexec` or `mpirun`.

Step 1: Communicators and Optimizers

In the following, we explain how to modify your code using Chainer to enable distributed training with ChainerMN. We take Chainer's MNIST example and modify it in a step-by-step manner to see the standard way of using ChainerMN.

Creating a Communicator

We first need to create a *communicator*. A communicator is in charge of communication between workers. A communicator can be created as follows:

```
comm = chainermn.create_communicator()
```

Workers in a node have to use different GPUs. For this purpose, `intra_rank` property of communicators is useful. Each worker in a node is assigned a unique `intra_rank` starting from zero. Therefore, it is often convenient to use the `intra_rank`-th GPU.

The following line of code is found in the original MNIST example:

```
chainer.cuda.get_device(args.gpu).use()
```

which we modify as follows:

```
device = comm.intra_rank
chainer.cuda.get_device(device).use()
```

Creating a Multi-Node Optimizer

This is the most important step. We need to insert the communication right after backprop and right before optimization. In ChainerMN, it is done by creating a *multi-node optimizer*.

Method `create_multi_node_optimizer` receives a standard Chainer optimizer, and it returns a new optimizer. The returned optimizer is called multi-node optimizer. It behaves exactly same as the supplied original standard optimizer (e.g., you can add hooks such as `WeightDecay`), except that it communicates model parameters and gradients properly in a multi-node setting.

The following is the code line found in the original MNIST example:

```
optimizer = chainer.optimizers.Adam()
```

To obtain a multi-node optimizer, we modify that part as follows:

```
optimizer = chainermn.create_multi_node_optimizer(
    chainer.optimizers.Adam(), comm)
```

Run

With the above two changes, your script is ready for distributed training. Invoke your script with `mpiexec` or `mpirun` (see your MPI's manual for details). The following is an example to execute the training with four processes at localhost:

```
$ mpiexec -n 4 python train_mnist.py
```

Multi-node execution

If you can successfully run the multi-process version of the MNIST example, you are almost ready for multi-node execution. The simplest way is to specify the `--host` argument to the **`mpiexec`** command. Let's suppose you have two GPU-equipped computing nodes: `host00` and `host01`, each of which has 4 GPUs, and so you have 8 GPUs in total:

```
$ mpiexec -n 8 -host host00,host01 python train_mnist.py
```

The script should print similar results to the previous intra-node execution.

Copying datasets

In the MNIST example, the rank 0 process reads the entire portion of the dataset and scatters it to other processes. In some applications, such as the ImageNet ChainerMN exmaple, however, only the pathes to each data file are scattered and each process reads the actual data files. In such cases, all datasets must be readable on all computing nodes in the same location. You don't need to worry about this if you use NFS (Network File System) or any other similar data synchronizing system. Otherwise, you need to manually copy data files between nodes using **`scp`** or **`rsync`**.

If you have trouble

If you have any trouble running the sample programs in your environment, go to the *Step-by-Step Troubleshooting* page and follow the steps to check your environment and configuration.

Next Steps

With only the above two changes distributed training is already performed. Thus, the model parameters are updated by using gradients that are aggregated over all the workers. However, this MNIST example still has a few areas in need of improvment. In the next page, we will see how to address the following problems:

- Training period is wrong; 'one epoch' is not one epoch.
- Evaluation is not parallelized.
- Status outputs to stdout are repeated and annoying.

Step 2: Datasets and Evaluators

Following from the previous step, we continue to explain general steps to modify your code for ChainerMN through the MNIST example. All of the steps below are optional, although useful for many cases.

Scattering Datasets

If you want to keep the definition of 'one epoch' correct, we need to scatter the dataset to all workers.

For this purpose, ChainerMN provides a method `scatter_dataset`. It scatters the dataset of worker 0 (i.e., the worker whose `comm.rank` is 0) to all workers. The given dataset of other workers are ignored. The dataset is split into sub datasets of almost equal sizes and scattered to the workers. To create a sub dataset, `chainer.datasets.SubDataset` is used.

The following line of code from the original MNIST example loads the dataset:

```
train, test = chainer.datasets.get_mnist()
```

We modify it as follows. Only worker 0 loads the dataset, and then it is scattered to all the workers:

```
if comm.rank == 0:
    train, test = chainer.datasets.get_mnist()
else:
    train, test = None, None

train = chainermn.scatter_dataset(train, comm)
test = chainermn.scatter_dataset(test, comm)
```

Replacing Epoch Triggers

This step is necessary only when you use `scatter_dataset`. Please remember that *using normal epoch triggers is dangerous*. This is because, when the length of the original dataset before scatter is not divisible by the number of workers, different workers may have sub datasets of different lengths. Therefore, epoch triggers may be invoked in different timings, and this may cause critical problems.

For this purpose, we offer a utility function `get_epoch_trigger`. Please note that this function communicates between workers, so, if you use it, then all the workers should call this.

The following line of code in the original MNIST example creates a trainer:

```
trainer = training.Trainer(updater, (args.epoch, 'epoch'), out=args.out)
```

We replace the stop trigger from an epoch trigger to the roughly same interval trigger by using `get_epoch_trigger` as follows:

```
trainer = training.Trainer(updater,
    chainermn.get_epoch_trigger(args.epoch, train, args.batchsize, comm),
    out=args.out)
```

Creating A Multi-Node Evaluator

This step is also an optional step, but useful when validation is taking a considerable amount of time. In this case, you can also parallelize the validation by using *multi-node evaluators*.

Similarly to multi-node optimizers, you can create a multi-node evaluator from a standard evaluator by using method `create_multi_node_evaluator`. It behaves exactly the same as the given original evaluator except that it reports the average of results over all workers.

The following line from the original MNIST example adds an evalautor extension to the trainer:

```
trainer.extend(extensions.Evaluator(test_iter, model, device=args.gpu))
```

To create and use a multi-node evaluator, we modify that part as follows:

```
evaluator = extensions.Evaluator(test_iter, model, device=device)
evaluator = chainermn.create_multi_node_evaluator(evaluator, comm)
trainer.extend(evaluator,
      trigger=chainermn.get_epoch_trigger(1, train, args.batchsize, comm))
```

Suppressing Unnecessary Extensions

Some of extensions should be invoked only by one of the workers. For example, if the `PrintReport` extension is invoked by all of the workers, many redundant lines will appear in your console. Therefore, it is convenient to register these extensions only at workers of rank zero as follows:

```
if comm.rank == 0:
    trainer.extend(extensions.dump_graph('main/loss'))
    trainer.extend(extensions.LogReport())
    trainer.extend(extensions.PrintReport(
        ['epoch', 'main/loss', 'validation/main/loss',
         'main/accuracy', 'validation/main/accuracy', 'elapsed_time']))
    trainer.extend(extensions.ProgressBar())
```

Please note that `chainermn.get_epoch_trigger` should not be used for these extensions, even if you use `scatter_dataset`. You should use normal epoch triggers because these extensions are activated only in one worker.

Tips and FAQs

Using MultiprocessIterator

If you are using `MultiprocessIterator` and communication goes through InfiniBand, you would probably face crashing problems. This is because `MultiprocessIterator` creates child processes by the `fork` system call, which has incompatibilities with the design of MPI and InfiniBand. To cope with this issue, we can use `multiprocessing.set_start_method` to change the way to start child processes:

```
multiprocessing.set_start_method('forkserver')
```

Both `forkserver` mode and `spawn` mode should work. Please also refer to our ImageNet example, where `MultiprocessIterator` is used. Unfortunately, `multiprocessing.set_start_method` is only available in Python 3.4+. Therefore you need those recent Python versions to use `MultiprocessIterator`.

Using Your Own Evaluator

Method `create_multi_node_evaluator` can also be used for customized evaluator classes that inherit from `chainer.training.extensions.Evaluator`. Specifically, it wraps the `evaluate` method and returns the averaged values over all workers. Please also refer to our ImageNet example, where a customized evaluator is used.

Using MPI4py Communicator

ChainerMN is based on MPI4py. For advanced users (e.g., those who want to parallelize preprocessing, create custom extension, etc.), we encourage you to make use of MPI4py communicators. Let `comm` be a ChainerMN communicator, then you can obtain MPI4py communicator by `comm.mpi_comm`. Please refer to MPI4py API reference.

Using FP16

FP16 (16-bit half precision floating point values) is not supported in ChainerMN as of now.

CHAPTER 3

API Reference

Communicators

chainermn.**create_communicator**(*communicator_name='hierarchical'*, *mpi_comm=None*)

Create a ChainerMN communicator.

Different communicators provide different approaches of communication, so they have different performance charasteristics. The default communicator `hierarchical` is expected to generally perform well on a variety of environments, so one need not to change communicators in most cases. However, choosing proper communicator may give better performance. The following communicators are available.

Name	CPU	GPU	NCCL	Recommended Use Cases
naive	OK	OK		Testing on CPU mode
hierarchical		OK	Required	Each node has a single NIC or HCA
two_dimensional		OK	Required	Each node has multiple NICs or HCAs
single_node		OK	Required	Single node with multiple GPUs
flat		OK		N/A

Parameters

- **communicator_name** – The name of communicator (`naive`, `flat`, `hierarchical`, `two_dimensional`, or `single_node`)
- **mpi_comm** – MPI4py communicator

Returns ChainerMN communicator

Optimizers and Evaluators

chainermn.**create_multi_node_optimizer**(*actual_optimizer*, *communicator*)

Create a multi node optimizer from a Chainer optimizer.

Parameters

- **actual_optimizer** – Chainer optimizer (e.g., `chainer.optimizers.Adam`).
- **communicator** – ChainerMN communicator.

Returns The multi node optimizer based on `actual_optimizer`.

`chainermn.`**create_multi_node_evaluator**(*actual_evaluator*, *communicator*)
Create a multi node evaluator from a normal evaluator.

Parameters

- **actual_evaluator** – evaluator (e.g., `chainer.training.extensions.Evaluator`)
- **communicator** – ChainerMN communicator

Returns The multi node evaluator based on `actual_evaluator`.

Dataset Utilities

`chainermn.`**scatter_dataset**(*dataset*, *comm*)
Scatter the given dataset to the workers in the communicator.

The dataset of worker 0 (i.e., the worker whose `comm.rank` is 0) is scattered to all workers. The given dataset of other workers are ignored. The dataset is split to sub datasets of almost equal sizes and scattered to workers. To create a sub dataset, `chainer.datasets.SubDataset` is used.

Parameters

- **dataset** – A dataset (e.g., `list`, `numpy.ndarray`, `chainer.datasets.TupleDataset`, ...).
- **comm** – ChainerMN communicator or MPI4py communicator.

Returns Scattered dataset.

`chainermn.`**get_epoch_trigger**(*n_epochs*, *dataset*, *local_batch_size*, *comm*)
Get the trigger that behaves like an epoch trigger.

Parameters

- **n_epochs** (*int*) – The number of epochs.
- **dataset** – Sub dataset of each worker.
- **local_batch_size** (*int*) – Batch size of each worker.
- **comm** – ChainerMN communicator or MPI4py communicator.

Returns The trigger that behaves like the epoch trigger.

`chainermn.`**get_n_iterations_for_one_epoch**(*dataset*, *local_batch_size*, *comm*)
Get the number of iterations for one epoch.

Parameters

- **dataset** – Sub dataset of each worker.
- **local_batch_size** (*int*) – Batch size of each worker.
- **comm** – ChainerMN communicator or MPI4py communicator.

Returns the number of iterations for one epoch.

Return type int

Python Module Index

C

chainermn, 11

Index

C

chainermn (module), 1, 11, 18
create_communicator() (in module chainermn), 19
create_multi_node_evaluator() (in module chainermn), 20
create_multi_node_optimizer() (in module chainermn), 19

E

environment variable
    LD_LIBRARY_PATH, 8
    MV2_SMP_USE_CMA, 8, 11
    MV2_USE_CUDA, 8, 11
    PATH, 8

G

get_epoch_trigger() (in module chainermn), 20
get_n_iterations_for_one_epoch() (in module chainermn), 20

L

LD_LIBRARY_PATH, 8

M

MV2_SMP_USE_CMA, 8, 11
MV2_USE_CUDA, 8, 11

P

PATH, 8

S

scatter_dataset() (in module chainermn), 20