ChainerMN Documentation

Release 1.3.1

Takuya Akiba

Oct 23, 2018

Contents

CHAPTER 1

Installation

1.1 Installation Guide

1.1.1 Requirements

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

NCCL

To enable efficient intra- and inter-node GPU-to-GPU communication, we use NVIDIA Collective Communications Library (NCCL). See the official instructions for installation.

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.

Note: We reccomend NCCL 2 but NCCL 1 can be used. When you use CUDA 7.0 and 7.5, please install NCCL 1 because NCCL 2 is not supported with CUDA 7.0 and 7.5. However, for NCCL 1, `PureNcclCommunicator` is not supported in ChainerMN. If you use NCCL 1, 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
```

If you change the version of NCCL installed, you have to reinstall CuPy. Because, current ChainerMN applies CuPy to use NCCL. See the official instructions for reinstalltion.

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.

CuPy

Chainer and ChainerMN rely on CuPy to use GPUs. Please refer to the official instructions to install. CuPy requires NCCL to be enabled. See *Check if NCCL is enabled in CuPy*, if you want to check whether NCCL is enabled in CuPy.

Chainer and ChainerMN can be installed without CuPy, in which case the corresponding features are not available. See *Non-GPU environments* for more details.

Tested Environments

We tested ChainerMN on all the following environments.

- OS
  - Ubuntu 14.04 LTS 64bit
  - Ubuntu 16.04 LTS 64bit

- Python 2.7.13 3.5.1 3.6.1
- Chainer 3.5.0 4.4.0
- CuPy 2.5.0 4.4.0
- MPI
  - openmpi 1.10.7 2.1.2
- MPI4py 3.0.0
- NCCL 2.2.13

1.1.2 Installation

Install via pip

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

```
$ pip install chainermn
```

NOTE: If you need **sudo** to use pip, you should be careful about environment variables. The **sudo** command DOES NOT inherit the environment, and thus you need to specify the variables explicitly.

```
$ sudo CPATH=${CPATH} LIBRARY_PATH=${LIBRARY_PATH} 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

Users who want to try ChainerMN in CPU-only environment may skip installation of CuPy. Non-GPU set up may not be performant as GPU-enabled set up, but would be useful for testing or debugging training program in non-GPU environment such as laptops or CI jobs.

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

Note: Current version of ChainerMN does not need `--no-nccl` flag for CPU-only environment at installation any more. It would be just ignored.

1.2 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.

1.2.1 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>
```

(continues on next page)

(continued from previous page)

```
#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)
```

Check if NCCL is enabled in CuPy

CuPy requires NCCL to be enabled. You can check it with the following command.:

```
$ python -c 'from cupy.cuda import nccl'
```

If you get an output like below, NCCL is not enabled in CuPy. Please check the installation guide of CuPy.:

```
Traceback (most recent call last):

 File "<string>", line 1, in <module>
```

(continues on next page)

(continued from previous page)

```
ImportError: cannot import name 'nccl'
```

1.2.2 Multi-node environment

Check SSH connection and environment 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):
```

(continues on next page)

(continued from previous page)

```
  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:
```

(continues on next page)

(continued from previous page)

```
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 environment 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 environment variables*)

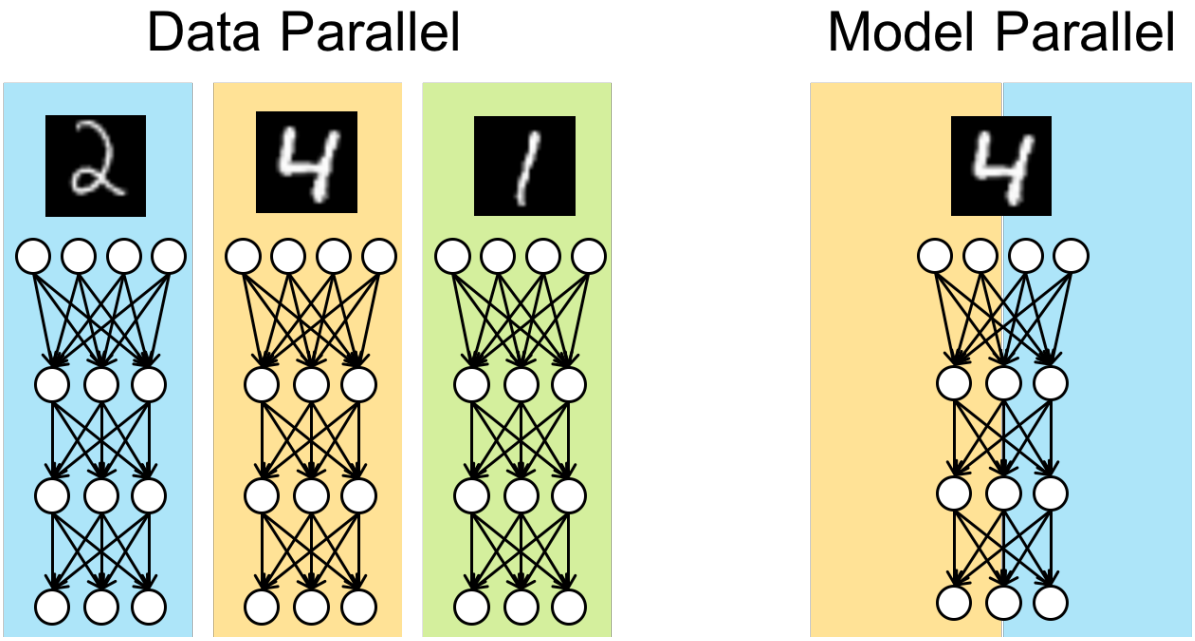
CHAPTER 2

Tutorial

2.1 Overview

2.1.1 Data Parallelism

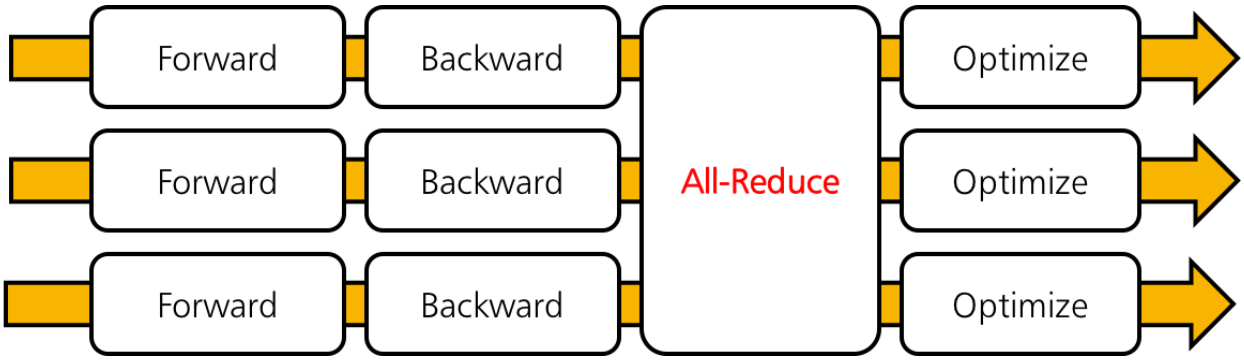
ChainerMN employs the data parallel approach for distributed training. In the data parallel approach, each worker has a model copy, and computes a gradient against a batch. Then, the workers collaborate to update the model using the gradients of all workers.



2.1.2 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.



2.1.3 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`.

2.2 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.

2.2.1 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_from_id(args.gpu).use()
```

which we modify as follows:

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

2.2.2 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)
```

2.2.3 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 of executing the training with four processes at localhost:

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

In the non-GPU mode, you may see a warning like shown below, but this message is harmless, and you can ignore it for now

```
Warning: using naive communicator because only naive supports CPU-only execution
```

If you have multiple GPUs on the localhost, 4 for example, you may also want to try:

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

2.2.4 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.

2.2.5 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 example, 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`**.

2.2.6 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.

2.2.7 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.

2.3 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.

2.3.1 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)
```

2.3.2 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 evaluator 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)
```

2.3.3 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())
```

2.4 Tips and FAQs

2.4.1 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, use `multiprocessing.set_start_method` to start child processes, with a process explicitly forked right after, **before communicator is created** as follows:

```
multiprocessing.set_start_method('forkserver')
p = multiprocessing.Process(target=lambda *x: x, args=())
p.start()
p.join()

communicator = chainermn.create_communicator(...)
```

Either `forkserver` mode or `spawn` mode should work. See our ImageNet example script for working sample code of `MultiprocessIterator` and `forkserver`. Unfortunately, `multiprocessing.set_start_method` is only available in Python 3.4+.

2.4.2 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.

2.4.3 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.

2.4.4 Using FP16

FP16 (16-bit half precision floating point values) is supported in `pure_nccl` of a ChainerMN communicator.

2.4.5 MPI process hangs after an unhandled Python exception.

An MPI runtime is expected to kill all of its child processes if one of them exits abnormally or without calling *MPI_Finalize()*. However, when a Python program runs on *mpi4py*, the MPI runtime often fails to detect the process failure, and the rest of the processes hang infinitely. It is especially problematic when you run your ChainerMN program on a cloud environment, in which you are charged on time basis.

This tiny program demonstrates the issue (note that it is not specific to ChainerMN).:

```
# test.py
def func():
  import mpi4py.MPI
  mpi_comm = mpi4py.MPI.COMM_WORLD
  if mpi_comm.rank == 0:
    raise ValueError('failure!')

  mpi4py.MPI.COMM_WORLD.Barrier()

if __name__ == '__main__':
  func()

# mpiexec -n 2 python test.py
```

mpi4py offers a solution to force all processes to abort if an uncaught exception occurs..

```
$ mpiexec -n 2 python -m mpi4py yourscript.py ...
```

This also works well with ChainerMN. See here for more details.

If you cannot apply the solution (i.e. you don't have a control of how Python interpreter is invoked), you can inject the following code snippet into your script file

```
import sys

# === begin code snippet
_old_hook = sys.excepthook

# Global error handler
def global_except_hook(exctype, value, traceback):
  import sys
  try:
      import mpi4py.MPI

$ mpiexec -n 2 -x CHAINERMN_FORCE_ABORT_ON_EXCEPTION=1 python yourscript.py ...
```

Alternatively, you can explicitly call `chainermn.global_except_hook.add_hook()` from your code.:

```
import chainermn

chainermn.global_except_hook.add_hook()
```

The handler hooks uncaught exceptions and call *MPI_Abort()* to ensure that all process are terminated.

You can choose any of these solutions depending on your environment and restrictions.

NOTE: These techniques are effective only for unhandled Python exceptions. If your program crashes due to lower-level issues such as *SIGSEGV*, the MPI process may still hang.

CHAPTER 3

API Reference

3.1 Communicators

chainermn.**create_communicator**(*communicator_name='hierarchical'*, *mpi_comm=None*, *allreduce_grad_dtype=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
pure_nccl		OK	Required (>= v2)	`pure_nccl` is recommended when NCCL2 is available in the environment.
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
naive	OK	OK		Testing on CPU mode

Parameters

- **communicator_name** – The name of communicator (`naive`, `flat`, `hierarchical`, `two_dimensional`, `pure_nccl`, or `single_node`)
- **mpi_comm** – MPI4py communicator
- **allreduce_grad_dtype** – Data type of gradient used in All-Reduce. If `None`, the dtype of a model is used.

Returns ChainerMN communicator that implements methods defined in *chainermn.CommunicatorBase*

class chainermn.**CommunicatorBase**

Interface definition of all communicators.

All communicators that have compatible set of methods with this class is supposed to work in ChainerMN's parallel computation implementation. The methods are named after MPI functions, such as `bcast()` came from `MPI_Bcast()`.

There are two types of methods: one that treats Python objects have `_obj` suffix. The other has methods without any suffix and it handles ndarray and arrays filled with scaler values. So the number of methods would be

```
[send, recv, bcast, gather, allreduce] * [ '_obj', '']
```

(with single exception `alltoall`, `allreduce_grad`, `split` and `bcast_data` so far). Also methods are supposed to be written in this order. All those methods must be implemented in its implementation class, or otherwise it cannot be instantiated in runtime.

Note: As most implementation of `_obj`-sufficed methods involves Python object pickling and unpickling, there is an implicit size limit.

TODO(kuenishi): as of now no implementation class actually has `allreduce` method.

allreduce(*data*)

Allreduce operation among processes

Processes one of several aggregation operations using all data from all processes and returns the result of the aggregation to all processes.

TODO(kuenishi): add `op` argument once we find a use case for operations other than 'SUM'.

Parameters **data** (*ndarray*) – the data to aggregate among all nodes.

Returns Sum of all data from all processes.

allreduce_grad(*model*)

Works as same as `allreduce_obj` but for Chainer model gradients

Note: this only supports *SUM* same as `allreduce_obj`.

---

allreduce_obj(*obj*)

Apply a reduce operation to all objects and spread the result.

For example of integers and summation, equivalent local code is:

```
>>> from functools import reduce
>>> reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])
15
```

The only operation currently supported is summation.

TODO(kuenishi): support other operations such as 'MAX', 'MIN' and 'PROD' with `op` argument once we need any of them.

Parameters **obj** – An arbitrary object to apply reduce operation. Must have corresponding operation method e.g. `__plus__()`.

Returns The result of the operation applied to all objects.

alltoall(*xs*)

All-to-all implementation for ndarray

Parameters **xs** (*tuple of numpy/cupy array*) –

Returns Received arrays. The length of tuple equals to the communicator size.

Return type ys (tuple of numpy/cupy array)

bcast (*data*, *max_buf_len=None*, *root=0*)
Broadcasts an ndarray from root process to all processes

Parameters

- **data** (*numpy/cupy array*) – for root process, the data to broadcast. For non-root processes, this argument is ignored.
- **max_buf_len** (*int*) – Length of send buffer.
- **root** (*int*) – the process who has the data to broadcast.

Returns The data sent from root process

Return type ys (numpy/cupy array)

bcast_data (*model*)
Broadcast Chainer model parameter data

bcast_obj (*obj*, *max_buf_len=None*, *root=0*)
Broadcasts an arbitrary object from root to all non-root processes.

Parameters

- **obj** – arbitrary object to broadcast to all other non-root processes. Will be ignored at all non-root processes.
- **max_buf_len** (*int*) – max length of the send buffer
- **root** (*int*) – rank of the root processes who sends an object

Returns an object sent from the root process.

gather (*data*, *root=0*)
Gathers an ndarray from all processes to root process

Parameters

- **data** (*ndarray, or scaler*) – for root process this is ignored. For For non-root processes, the data to send to root process.
- **root** (*int*) – rank of the process who receives the data.

Returns For root process, the ndarray sent from non-root processes. For non-root processes, what?

gather_obj (*obj*, *root=0*)
Gathers arbitrary objects from all non-root processes to root process.

Parameters

- **obj** – arbtrary object to send to root process. Root process will receive this argument included in returned list.
- **root** (*int*) – rank of the root node who receives all objects.

Returns A list of objects sent from all processes.

TODO(kuenishi): make sure the ordering of objects in the returned list.

inter_rank
The rank of this node in the cluster.

inter_size
Number of nodes that participates the cluster.

intra_rank
Intra rank (process id in the machine) of this process.

rank
Rank (process id in the cluster) of this process in integer.

recv(*source*, *tag*)
Receives an ndarray from source.

To receive the message, sender must send the data.

Parameters

- **source** (*int*) – Rank of the source process
- **tag** (*int*) – The tag to specifically receive the message

Returns The data sent from source process

recv_obj(*source*, *tag*)
Receives an arbitrary Python object from source process with a tag.

Parameters

- **source** (*int*) – Rank number of sender process, to selectively receive the object.
- **tag** – tag to identify the message.

Returns an object sent from the source by `send_obj`.

send(*data*, *dest*, *tag*)
Sends an ndarray to destination

Receiver must invoke `recv()` to wait for the message.

Parameters

- **data** – data to be sent (tuple, list or raw numpy/cupy array)
- **dest** (*int*) – Rank of the destination process
- **tag** (*int*) – The tag to identify the message

send_obj(*obj*, *dest*, *tag*)
Sends an arbitrary Python object to destination with a tag.

Parameters

- **obj** – Arbitrary object to send to receiver.
- **dest** (*int*) – Rank number of receiver process (destination).
- **tag** – tag to identify the message.

size
Number of processes of the cluster.

split(*color*, *key*)
A function anologous to `MPI_Comm_Split` .

This method splits the inter MPI commnicator and return a wrapped ChainerMN communicator.

Parameters

- **color** (*int*) – Index of new group. The process with the same color will be assigned to the same group.
- **key** (*int*) – Control of rank assignment. The process will be assigned a rank in the new group ordered by the value of key. If you do not care of the rank, you can just simply specify the original rank.

Returns CommunicatorBase

3.2 Optimizers and Evaluators

chainermn.**create_multi_node_optimizer**(*actual_optimizer*, *communicator*, *double_buffering=False*)

Create a multi node optimizer from a Chainer optimizer.

Parameters

- **actual_optimizer** – Chainer optimizer (e.g., `chainer.optimizers.Adam`).
- **communicator** – ChainerMN communicator.
- **double_buffering** – If `True`, all-reduce and other processing (such as forward and backward) are overlapped using double buffering. There are cases where accuracy is affected because the gradients of the previous iteration are used for update. This flag is supported by `PureNcclCommunicator` only.

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.

Actually this method patches the evaluator to work in multi node environment. This method adds several hidden attributes starting with *_mn_* prefix.

Parameters

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

Returns The multi-node patched `actual_evaluator`.

Note: After patched, original evaluator does not work correctly in non-MPI environment.

3.3 Dataset Utilities

chainermn.**scatter_dataset**(*dataset*, *comm*, *root=0*, *shuffle=False*, *seed=None*, *max_buf_len=268435456*)

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.
- **shuffle** (*bool*) – If `True`, the order of examples is shuffled before being scattered.
- **root** (*int*) – The root process of the scatter operation.
- **seed** (*int*) – Seed the generator used for the permutation of indexes. If an integer being convertible to 32 bit unsigned integers is specified, it is guaranteed that each sample in the given dataset always belongs to a specific subset. If `None`, the permutation is changed randomly.
- **max_buf_len** (*int*) – Max buffer size to be used at broadcasting binaries. Must not be larger than 2147483647.

Returns Scattered dataset.

`chainermn.datasets.`**create_empty_dataset**(*dataset*)

Creates an empty dataset for models with no inputs and outputs.

This function generates an empty dataset, i.e., `__getitem__()` only returns `None`. Its dataset is compatible with the original one. Such datasets used for models which do not take any inputs, neither return any outputs. We expect models, e.g., whose `forward()` is starting with `chainermn.functions.recv()` and ending with `chainermn.functions.send()`.

Parameters **dataset** – Dataset to convert.

Returns Dataset consists of only patterns in the original one.

Return type TransformDataset

3.4 Links

class `chainermn.`**MultiNodeChainList**(*comm*)

Combining multiple non-connected components of computational graph.

This class combines each `chainer.Chain`, which represents one of the non-connected component in compuational graph. In `__call__()`, the returned object of `chainer.Chain` (which represents pointer) are passed to the next `chainer.Chain`, in order to retain the computational graph connected and make backprop work properly.

Users add each `chainer.Chain` by `add_link()` method. Each chain is invoked in forward computation according to the order they are added, and in backward computation according to the reversed order.

Example (basic usage)

This is a simple example of the model which sends its outputs to rank=1 machine:

```
import chainer
import chainer.functions as F
import chainermn


class SimpleModelSub(chainer.Chain):

    def __init__(self, n_in, n_hidden, n_out):
        super(SimpleModelSub, self).__init__(
```

(continues on next page)

(continued from previous page)

```
            l1=L.Linear(n_in, n_hidden),
            l2=L.Linear(n_hidden, n_out))

    def __call__(self, x):
        h1 = F.relu(self.l1(x))
        return self.l2(h1)


class SimpleModel(chainermn.MultiNodeChainList):

    def __init__(self, comm, n_in, n_hidden, n_out):
        super(SimpleModel, self).__init__(comm)
        self.add_link(
            SimpleModelSub(n_in, n_hidden, n_out),
            rank_in=None,
            rank_out=1)
```

Example (split MLP on 2 processes)

This is the other example of two models interacting each other:

```
import chainer
import chainer.functions as F
import chainermn


class MLP(chainer.Chain):

    def __init__(self, n_in, n_hidden, n_out):
        super(MLP, self).__init__(
            l1=L.Linear(n_in, n_hidden),
            l2=L.Linear(n_hidden, n_hidden),
            l3=L.Linear(n_hidden, n_out))

    def __call__(self, x):
        h1 = F.relu(self.l1(x))
        h2 = F.relu(self.l2(h1))
        return self.l3(h2)


class Model0(chainermn.MultiNodeChainList):

    def __init__(self, comm):
        super(Model0, self).__init__(comm)
        self.add_link(
            MLP(10000, 5000, 2000),
            rank_in=None,
            rank_out=1)
        self.add_link(
            MLP(100, 50, 10),
            rank_in=1,
            rank_out=None)


class Model1(chainermn.MultiNodeChainList):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, comm):
        super(Model1, self).__init__(comm)
        self.add_link(MLP(2000, 500, 100), rank_in=0, rank_out=0)
```

`Model0` is expected to be on rank=0, and `Model1` is expected to be on rank=1. The first `MLP` in `Model0` will send its outputs to `Model1`, then `MLP` in `Model1` will receive it and send its outputs to the second `MLP` in `Model0`.

Example (sending tuples)

This is the example for sending a tuple:

```
import chainer
import chainer.functions as F
import chainermn

class NN0(chainer.Chain):
    def __call__(self, x):
        y0 = some_calculation_nn0_0(x)
        y1 = some_calculation_nn1_1(x)
        return y0, y1

class NN1(chainer.Chain):
    def __call__(self, y):
        y0, y1 = y  # unpack tuple from NN0
        return some_calculation_nn1(y0, y1)

class Model_on_Process_0(chainermn.MultiNodeChainList):
    def __init__(self, comm):
        super(Model_on_Process_0, self).__init__(comm=comm)
        self.add_link(NN0(), rank_in=None, rank_out=1)

class Model_on_Process_1(chainermn.MultiNodeChainList):
    def __init__(self, comm):
        super(Model_on_Process_1, self).__init__(comm=comm)
        self.add_link(NN1(), rank_in=0, rank_out=None)
```

In this example, `Model_on_Process_0` sends two elemental tuple `(y0, y1)` (returned by `NN0.__call__`) to `Model_on_Process_1`, which can be unpacked as shown in `NN1.__call__`.

Parameters **comm** (*chainermn.communicators._base.CommunicatorBase*) – ChainerMN communicator.

add_link(*link*, *rank_in=None*, *rank_out=None*)
Register one connected link with its inout rank.

Parameters

- **link** (*chainer.Link*) – The link object to be registered.
- **rank_in** (*int, list, or None*) – Ranks from which it receives data. If None is specified, the model does not receive from any machines.

- **rank_out** (*int, list, or None*) – Ranks to which it sends data. If None is specified, the model will not send to any machine.

class chainermn.links.**MultiNodeBatchNormalization**(*size*, *comm*, *decay=0.9*, *eps=2e-05*, *dtype=<type 'numpy.float32'>*, *use_gamma=True*, *use_beta=True*, *initial_gamma=None*, *initial_beta=None*, *communication_backend='auto'*)

Batch normalization layer that can use the whole batch stats.

When using chainer.link.BatchNormalization, batch mean and std are computed independently for the local batch in each worker. When local batch size is too small, training is unstable due to unreliable batch stats.

In contrast, when using this MultiNodeBatchNormalization, workers communicate to conduct 'correct' batch normalization (e.g., obtaining mean and std for the whole global batch).

This link works only with Chainer >= 2.0.0.

Parameters

- **size** (*int or tuple of ints*) – Size (or shape) of channel dimensions.
- **comm** (*ChainerMN communicator*) – communicator to share the batch stats.
- **decay** (*float*) – Decay rate of moving average. It is used on training.
- **eps** (*float*) – Epsilon value for numerical stability.
- **dtype** (*numpy.dtype*) – Type to use in computing.
- **use_gamma** (*bool*) – If `True`, use scaling parameter. Otherwise, use unit(1) which makes no effect.
- **use_beta** (*bool*) – If `True`, use shifting parameter. Otherwise, use unit(0) which makes no effect.
- **communication_backend** (*str*) – `mpi`, `nccl` or `auto`. It is used to determine communication backend. If `auto`, use the best communication backend for each communicator.

3.5 Functions

chainermn.functions.**send**(*x*, *communicator*, *rank*, *tag=0*)

Send elements to target process.

This function returns a dummy variable only holding the computational graph. If `backward()` is invoked by this dummy variable, it will try to receive gradients from the target process and send them back to the parent nodes.

Parameters

- **x** (*Variable*) – Variable holding a matrix which you would like to send.
- **communicator** (*chainer.communicators.CommunicatorBase*) – ChainerMN communicator.
- **rank** (*int*) – Target process specifier.
- **tag** (*int*) – Optional message ID (MPI feature).

Returns A dummy variable with no actual data, only holding the computational graph. Please refer `chainermn.functions.pseudo_connect` for detail.

Return type Variable

chainermn.functions.**recv**(*communicator*, *rank*, *delegate_variable=None*, *tag=0*, *force_tuple=False*)

Receive elements from target process.

This function returns data received from target process. If `backward()` is invoked, it will try to send gradients to the target process. The received array will be on the current CUDA device if the corresponding `send()` is invoked with arrays on GPU. Please be aware that the current CUDA device is intended one. (`https://docs-cupy.chainer.org/en/stable/tutorial/basic.html#current-device`)

Note: If you define non-connected computational graph on one process, you have to use `delegate_variable` to specify the output of previous computational graph component. Otherwise `backward()` does not work well. Please refer `chainermn.functions.pseudo_connect` for detail.

Parameters

- **communicator** (*chainer.communicators.CommunicatorBase*) – ChainerMN communicator.
- **rank** (*int*) – Target process specifier.
- **delegate_variable** (*chainer.Variable*) – Pointer to the other non-connected component.
- **tag** (*int*) – Optional message ID (MPI feature).
- **force_tuple** (*bool*) – If `False` (the default) a Variable will be returned when the number of outputs is one. Otherwise, this method returns a tuple even when the number of outputs is one.

Returns Data received from target process. If `backward()` is invoked by this variable, it will send gradients to the target process.

Return type Variable

chainermn.functions.**pseudo_connect**(*delegate_variable*, **actual_variables*)

Connect independent connected graph component.

This function is implemented to return received arguments directly, except the first `delegate_variable`. In backward computation, it returns received gradients directly, adding a zero grad corresponding to `delegate_variable`. The detail of `delegate_variable` is described in the following notes.

Note: In model-parallel framework, models on each process might have many non-connected components. Here we call a given graph non-connected when multiple inter-process communications are needed for its computation. For example, consider the following example:

```
class ConnectedGraph(chainermn.MultiNodeChainList):

    def __init__(self, comm):
        super(ConnectedGraph, self).__init__(comm)
        self.add_link(ConnectedGraphSub(), rank_in=3, rank_out=1)
```

This model receives inputs from rank=3 process and sends its outputs to rank=1 process. The entire graph can be seen as one connected component `ConnectedGraphSub`. Please refer the document of `MultiNodeChainList` for detail.

On the other hand, see the next example:

```
class NonConnectedGraph(chainermn.MultiNodeChainList):

    def __init__(self, comm):
        super(NonConnectedGraph, self).__init__(comm)
        self.add_link(NonConnectedGraphSubA(), rank_in=3, rank_out=1)
        self.add_link(NonConnectedGraphSubB(), rank_in=1, rank_out=2)
```

This model consists of two components: at first, `NonConnectedGraphSubA` receives inputs from rank=3 process and sends its outputs to rank=1 process, and then `NonConnectedGraphSubB` receives inputs from rank=1 process and sends its outputs to rank=2 process. Here multiple inter-process communications are invoked between `NonConnectedGraphSubA` and `NonConnectedGraphSubB`, so it is regarded as non-connected.

Such kind of non-connected models can be problematic in backward computation. Chainer traces back the computational graph from the output variable, however naive implementation of `chainermn.functions.recv` does not take any inputs rather receives inputs by `MPI_Recv`, where backward path vanishes.

To prevent this, dummy variables what we call `delegate_variable` are used. In principle, `chainermn.functions.send` does not return any outputs because it sends data to the other process by `MPI_Send`. However, `chainermn.functions.send` returns a dummy / empty variable in our implementation, which is called `delegate_variable`. This variable does not hold any data, just used for retaining backward computation path. We can guarantee the backward computation just by putting `delegate_variable` to the next `chainermn.functions.recv` (`chainermn.functions.recv` has an optional argument to receive `delegate_variable`).

Note: In some cases the intermediate graph component returns model outputs. See the next example:

```
class NonConnectedGraph2(chainermn.MultiNodeChainList):

    def __init__(self, comm):
        super(NonConnectedGraph2, self).__init__(comm)
        self.add_link(NonConnectedGraphSubA(), rank_in=1, rank_out=None)
        self.add_link(NonConnectedGraphSubB(), rank_in=None, rank_out=1)
```

This model first receives inputs from rank=1 process and make model outputs (specified by `rank_out=None`) in `NonConnectedGraphSubA`. Then using model inputs (specified by `rank_in=None`), `NonConnectedGraphSubB` sends its outputs to rank=1 process. Since `MultiNodeChainList.__call__` returns outputs of the last component (in this case, outputs of `NonConnectedGraphSubB`), naive implementation cannot output the returned value of `NonConnectedGraphSubA` as the model outputs. In this case, `pseudo_connect` should be used.

`pseudo_connect` takes two arguments. The first one `delegate_variable` is what we explained in above note. In this case, returned value of `NonConnectedGraphSubB` corresponds to `delegate_variable`. The second one `actual_variables` is "what we want `delegate_variable` to imitate". In `NonConnectedGraph2`, we obtain returned value of `NonConnectedGraphSubB` as the model outputs, but what we actually want is returned value of `NonConnectedGraphSubA`. At the same time we want to trace back this resulted variable in backward computation. Using `pseudo_connect`, we can make a variable whose data is the same as the returned value of `NonConnectedGraphSubA`, and which traces back `NonConnectedGraphSubB` first.

`pseudo_connect` should also be used in some pathological cases, for example, where multiple `chainermn.functions.send` occurs sequentially.

Parameters

- **delegate_variable** (*chainer.Variable*) – Pointer to the previous non-connected graph component.
- **actual_variables** (*tuple of chainer.Variable*) – Actual values which delegate_variable imitate.

Returns A variable with the given values combined with delegating variable.

Return type Variable

chainermn.functions.**bcast**(*comm*, *x*, *root=0*)

Differentiable broadcast communication between workers.

This function invokes broadcast communications among processes specified by the communicator. Backward will be invoked as well as the ordinary chainer functions, where gradients are gathered to the root process and summed up.

The received array will be on the current CUDA device if x on the invoking process is on GPU. Please be aware that the current CUDA device is intended one. (https://docs-cupy.chainer.org/en/stable/tutorial/basic.html#current-device)

Parameters

- **comm** – ChainerMN communicator.
- **x** (*chainer.Variable*) – Variable to be sent.

Returns Broadcasted variable.

Return type y (chainer.Variable)

chainermn.functions.**gather**(*comm*, *x*, *root=0*)

Differentiable gather communication between workers.

This function invokes gather communications among processes specified by the communicator. Backward will be invoked as well as the ordinary chainer functions, where gradients are scattered from the root process to each slave.

The received array will be on the current CUDA device if x on the root process is on GPU. Please be aware that the current CUDA device is intended one. (https://docs-cupy.chainer.org/en/stable/tutorial/basic.html#current-device)

Parameters

- **comm** – ChainerMN communicator.
- **x** (*chainer.Variable*) – Variable to be sent.

Returns Gathered variables. None for slaves.

Return type ys (chainer.Variable)

chainermn.functions.**scatter**(*comm*, *xs*, *root=0*)

Differentiable scatter communication between workers.

This function invokes scatter communications among processes specified by the communicator. Backward will be invoked as well as the ordinary chainer functions, where gradients are gathered to the root process.

The received array will be on the current CUDA device if xs on the root process is on GPU. Please be aware that the current CUDA device is intended one. (https://docs-cupy.chainer.org/en/stable/tutorial/basic.html#current-device)

Parameters

- **comm** – ChainerMN communicator.

- **xs** (*list of chainer.Variable*) – Variables to be scattered for master process. None for slave process.

Returns Scattered variable.

Return type y (chainer.Variable)

chainermn.functions.**alltoall**(*comm*, *xs*)

Differentiable all-to-all communication between workers.

This function invokes all-to-all communications among processes specified by the communicator. Backward will be invoked as well as the ordinary chainer functions, just passing input gradients back. Unlike point-to-point communication such as chainermn.functions.send and chainermn.functions.recv, users need not to care about delegate variables, since backward() will not be invoked until all gradients from output direction arrive. Please refer to chainermn.functions.pseudo_connect about the detail of delegate variables.

The received array will be on the current CUDA device on the invoking process if xs is on GPU. Please be aware that the current CUDA device is intended one. (https://docs-cupy.chainer.org/en/stable/tutorial/basic.html#current-device)

Parameters

- **comm** – ChainerMN communicator.
- **xs** (*list of chainer.Variables*) – Variables to send.

Returns Received variables.

Return type ys (list of chainer.Variables)

chainermn.functions.**allgather**(*comm*, *x*)

Differentiable all-gather communication between workers.

This function invokes gather communications among processes specified by the communicator. Backward will be invoked as well as the ordinary chainer functions, where gradients are reduced to each process.

The received array will be on the current CUDA device on the invoking process if x is on GPU. Please be aware that the current CUDA device is intended one. (https://docs-cupy.chainer.org/en/stable/tutorial/basic.html#current-device)

Parameters

- **comm** – ChainerMN communicator.
- **x** (*chainer.Variables*) – Variables to send.

Returns Received variables.

Return type ys (list of chainer.Variables)

3.6 Iterators

chainermn.iterators.**create_multi_node_iterator**(*actual_iterator*, *communicator*, *rank_master=0*)

Create a multi node iterator from a Chainer iterator.

This iterator shares the same batches on multiple processes, simply broadcasting batches from master process to slave processes in each iteration. Master process obtains batches from actual_iterator, which you can specify any Chainer iterator (e.g. chainer.iterators.SerialIterator).

Here is an example situation. When we train a sequence-to-sequence model, where the encoder and the decoder is located on two different processes, we want to share the same batches on each process, thus inputs for the encoder and output teacher signals for the decoder become consistent.

In order to use the multi node iterator, first create the iterator from Chainer iterator and ChainerMN communicator:

```
iterator = chainermn.iterators.create_multi_node_iterator(
    chainer.iterators.SerialIterator(
        dataset, batch_size, shuffle=True),
    communicator)
```

Then you can use it as the ordinary Chainer iterator:

```
updater = chainer.training.StandardUpdater(iterator, optimizer)
trainer = training.Trainer(updater)
trainer.run()
```

Since this iterator shares batches through network in each iteration, communication might be large. If you train your model-parallel network on extremely large dataset, you can also consider to use `chainermn.iterators.create_synchronized_iterator`.

Current multi node iterator supports numpy.float32 or tuple of numpy.float32 as the data type of the batch element.

Note: `create_multi_node_iterator` and `serialize` of created iterators must be called at the same time by master and slaves, unless it falls into deadlock because they synchronize internal states of iterators.

Parameters

- **actual_iterator** – Chainer iterator (`chainer.iterators.SerialIterator` and `chainer.iterators.MultiprocessIterator` are supported).
- **communicator** – ChainerMN communicator.
- **rank_master** – process rank to be master.

Returns The master-slave iterator based on `actual_iterator`.

3.7 Trainer extensions

class chainermn.extensions.**AllreducePersistent**(*model*, *comm*)

Chainer extension to averagize persistents over workers.

When called, this extension invokes all-reduce communication among workers to compute averages of persistent variables in the model. Persistent variables are updated to the averages. Currently, we ignore integer persistent variables, and only float persistent variables are handled.

This extension is mainly to improve the running mean and variance of BatchNormalization by increasing the effective number of examples. We do not need to call this frequently; call just before storing or evaluating the model.

Parameters

- **model** (*chainer.link.Link*) – Target link object.
- **comm** (*ChainerMN communicator*) – communicator to compute averages.

chainermn.**create_multi_node_checkpointer**(*name*, *comm*, *cp_interval=5*, *gc_interval=5*, *path=None*)

Create multi-node checkpointer object

Generational snapshot extension to allow fault tolerance; It keeps several old snapshots to rollback synchronized snapshot at each MPI process. Snapshot files are identified as '<name>.<rank>.<iteration>'.

- <name> ... identifier of the run where snapshot is kept for
- <rank> ... which process owned the model
- <iteration> ... number of iteration.

This extension keeps several files for each execution and allows users to resume the whole job at the latest snapshots of each MPI process, and the iteration where all snapshots agrees.

As this object is a usual Chainer extension, users can just create this object and pass to the trainer as an extension:

```
checkpointer = create_multi_node_checkpointer(name=run_id, comm=comm)
trainer.extend(checkpointer, trigger=(25, 'iteration'))
```

To run recovery at startup, before first iteration, run

checkpointer.maybe_load(trainer, optimizer)

before `trainer.run()`. If nothing is recovered (i.e. no snapshot found), `trainer.updater.iteration` will remain `0`. Otherwise it will have the value of snapshot and the training will resume from that iteration. `optimizer` is optional but this will let multi node optimizer avoid initial broadcast when all snapshot data among nodes are all in sync.

Note: Make sure that `checkpointer.maybe_load` is called *after* all extensions with states, such as `ExponentialShift`, set to the trainer.

After training finished without errors all those temporary checkpoints will be cleaned up at all nodes.

Another example to use checkpointer *without* trainer would be:

```
checkpointer = create_multi_node_checkpointer(name=run_id, comm=comm)
checkpointer.maybe_load(obj_you_want_to_snap, optimizer)

while True: ## Training loop
    ...
    updater.update()
    ...
    checkpointer.save(obj_you_want_to_snap)  # Make a checkpoint
```

Parameters

- **name** (*str*) – unique id of the run
- **comm** – communicater in ChainerMN
- **cp_interval** (*int*) – minimum number of checkpoints to preserve
- **gc_interval** (*int*) – interval to collect non-preserved checkpoints

3.8 Configurations

3.8.1 Environmental Variables

CHAINERMN_FORCE_ABORT_ON_EXCEPTIONS If this variable is set to a non-empty value, ChainerMN installs a global hook to Python's *sys.excepthook* to call `MPI_Abort()` when an unhandled exception occurs. See faq-global-except-hook

ChainerMN issue #236 may also help to understand the problem.

3.8.2 Execution Control

`chainermn.global_except_hook.`**`add_hook`**`()`
Add a global hook function that captures all unhandled exceptions.

The function calls MPI_Abort() to force all processes abort. It is useful when you run your training script on a cloud platform.

Python Module Index

C

Index