EPL

Alibaba-inc

Mar 31, 2023

EPL

Easy Parallel Library (EPL) is a general and efficient deep learning framework for distributed giant model training.

CHAPTER

ONE

EASY PARALLEL LIBRARY

1.1 Overview

Easy Parallel Library (EPL) is a general and efficient library for distributed model training.

- Usability - Users can implement different parallelism strategies with a few lines of annotations, including data parallelism, pipeline parallelism, tensor model parallelism, and their hybrids.
- Memory Efficient - EPL provides various memory-saving techniques, including gradient checkpoint, ZERO, CPU Offload, etc. Users are able to train larger models with fewer computing resources.
- High Performance - EPL provides an optimized communication library to achieve high scalability and efficiency.

1.2 Examples

Here are a few examples of different parallelism strategies by changing only annotations. Please refer to API documentation for API details and tutorials for more examples.

1.2.1 Data Parallelism

The following example shows a basic data parallelism annotation. The data parallelism degree is determined by the allocated GPU number.

```
+ import epl
+ epl.init()
+ with epl.replicate(device_count=1):
    model()
```

1.2.2 Pipeline Parallelism

The following example shows pipeline parallelism with two pipeline stages, each stage is computed with one GPU. If the total GPU number is 4, EPL will automatically apply two-degree data parallelism over the model pipeline.

```
+ import epl
+
+ config = epl.Config({"pipeline.num_micro_batch": 4})
+ epl.init(config)
+ with epl.replicate(device_count=1, name="stage_0"):
```

(continues on next page)

(continued from previous page)

```
    model_part1()
+ with epl.replicate(device_count=1, name="stage_1"):
    model_part2()
```

1.2.3 Tensor Model Parallelism

The following example shows a tensor model parallelism annotation. We apply data parallelism to the `ResNet` part, and apply tensor model parallelism to `classification` part.

```
+ import epl
+ config = epl.Config({"cluster.colocate_split_and_replicate": True})
+ epl.init(config)
+ with epl.replicate(8):
    ResNet()
+ with epl.split(8):
    classification()
```

1.3 Publication

If you use EPL in your publication, please cite it by using the following BibTeX entry.

```
@misc{jia2021whale,
      title={Whale: Scaling Deep Learning Model Training to the Trillions},
      author={Xianyan Jia and Le Jiang and Ang Wang and Jie Zhang and Xinyuan Li and␣
↪Wencong Xiao and Langshi chen and Yong Li and Zhen Zheng and Xiaoyong Liu and Wei Lin},
      year={2021},
      eprint={2011.09208},
      archivePrefix={arXiv},
      primaryClass={cs.DC}
}
```

1.4 Contact Us

Feel free to open an issue, or join the Official Discussion Group on DingTalk.


EPL用户交流群
EPL
扫一扫群二维码，立刻加入该群。

CHAPTER

TWO

INSTALLATION

You can install EPL by following instructions.

2.1 Requirements

- TensorFlow-GPU 1.15

2.2 Install from source

2.2.1 Build from NVIDIA TF1.15 DOCKER

```
nvidia-docker run -ti --gpus all --name build_epl_with_nvtf1.15_21.12 --net host --ipc␣
↪host -v /mnt:/mnt nvcr.io/nvidia/tensorflow:21.12-tf1-py3 bash

# clone and install EPL
git clone https://github.com/alibaba/EasyParallelLibrary.git
cd EasyParallelLibrary
pip install .
```

2.2.2 Build from TensorFlow TF1.15 DOCKER

```
nvidia-docker run -ti --gpus all --name build_epl_with_tf1.15 --net host --ipc host -v /
↪mnt:/mnt tensorflow/tensorflow:1.15.5-gpu-py3 bash
# install nccl
apt update
apt install libnccl2 libnccl-dev

# clone and install EPL
git clone https://github.com/alibaba/EasyParallelLibrary.git
cd EasyParallelLibrary
pip install .
```

CHAPTER

THREE

QUICK START

In this section, we will use a simple DNN training example to show how to use EPL for distributed training.

3.1 EPL Annotation

A user needs to first annotate `local_model.py` with EPL parallelism strategies. The following example shows a data parallelism sample by adding three lines.

```
# local_model.py
import numpy as np
import tensorflow as tf
+ import epl

+ epl.init()
+ epl.set_default_strategy(epl.replicate(1))

# Define model
num_x = np.random.randint(0, 10, (500, 20)).astype(dtype=np.float32)
num_y = np.random.randint(0, 10, 500).astype(dtype=np.int64)
dataset = tf.data.Dataset.from_tensor_slices((num_x, num_y)).batch(10).repeat(1)
iterator = dataset.make_initializable_iterator()
tf.add_to_collection(tf.GraphKeys.TABLE_INITIALIZERS, iterator.initializer)
x, labels = iterator.get_next()
logits = tf.layers.dense(x, 2)
logits = tf.layers.dense(logits, 10)
loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)
global_step = tf.train.get_or_create_global_step()
optimizer = tf.train.MomentumOptimizer(learning_rate=0.001, momentum=0.9)
train_op = optimizer.minimize(loss, global_step=global_step)

# Training session
with tf.train.MonitoredTrainingSession() as sess:
  for i in range(10):
    train_loss, _, step = sess.run([loss, train_op, global_step])
    print("Iteration %s , Loss: %s ." % (step, train_loss))
print("Train Finished.")
```

3.2 Launch a parallel training

Then the user needs to provide a local launch script such as `run.sh`, as follows:

```
# run.sh
python local_model.py
```

The following script launches a parallel training program with 1 worker and 2 GPUs.

```
epl-launch --num_workers 1 --gpu_per_worker 2 run.sh
```

CHAPTER

FOUR

PARALLELISM STRATEGY API

In this section, we will introduce the parallelism primitive API, which can be used to build various parallelism strategies.

Firstly, we will recap some basic concepts used in this document.

- *Model replica*: local DNN model (without parallelism or gradient accumulation).
- *micro batch size(mb)*: number of samples consumed by one model replica in each training iteration.
- *num_micro_batch*: number of micro batch used in pipeline or GA for each model replica in each training iteration.
- *global batch size*: Assume the model replica number is N, then the global batch size is `N * mb * num_micro_batch`.
- *TaskGraph*: TaskGraph is a subset of the model for parallel transformation and execution.

Unless otherwise specified, the default batch size of the local model is `micro batch size`.

4.1 Parallel Strategy Primitive

With strategy primitive annotation, EPL partitions the model into multiple `TaskGraphs` and applies the parallelism strategies to the `TaskGraphs`. EPL provides two basic strategy primitives: `replicate` and `split`. Each strategy annotation generates one `TaskGraph`.

4.1.1 replicate

`replicate` annotates operations to data parallelism, where each replica consumes different input data. Operations defined under `replicate` scope form one `TaskGraph`.

1. If the whole model is annotated with `replicate`i.e. there is one `TaskGraph`, then it is the same as the traditional data parallelism.
2. If part of the model is annotated with `replicate`, EPL will perform data parallelism for the corresponding TaskGraph.

API definition:

```
replicate(device_count=None, name=None)
```

Args	Required	Description
device_count	True	device count for one model replica defined under `replicate` scope.
name		strategy name

For data parallelism, one model replica is placed in one GPU (`device_count=1`), and EPL will infer the total number of replicas given the allocated number of GPUs. When `device_count>1`, EPL will split the input batch into `device_count` parts when replicating the model, and keeps the total batch size of replicas the same as the original local batch size.

The following examples show data parallelism, where each model replica is placed in one GPU. If the total allocated GPU number is 8, then the model will be scaled to 8 GPUs to perform data parallelism training.

```
import epl
epl.init()
with epl.replicate(device_count=1):
  model()
```

4.1.2 split

`split` annotates model to be split. Operations defined under `split` scope form a `TaskGraph`, which is split over multiple GPUs for parallel computation.

API definition:

```
split(device_count=None, name=None)
```

Args	Required	Description
device_count	True	number of devices to split and place the model.
name		strategy name

The following example shows the tensor model parallelism. The model is split over 8 GPUs.

```
import epl
epl.init()
with epl.split(device_count=8):
  model()
```

4.2 set_default_strategy

EPL also provides `set_default_strategy` to set the default parallelism strategies for operations.

```
set_default_strategy(strategy)
```

Args	Required	Description
strategy	True	parallelism strategy.

The following example shows the data parallelism by setting the default strategy to `replicate`.

```
import epl
epl.init()
epl.set_default_strategy(epl.replicate(device_count=1))
model()
```

4.3 API Instruction

- By default, different TaskGraphs are placed in different devices.
- We do not allow nesting strategy annotations.
- Users only need to annotate the forward part of the model, the backward and apply operations are automatically co-located with the forward operations.

To learn how to use the above API to implement various parallelism strategies, such as pipeline parallelism or hybrid parallelism, please refer to *parallelism examples*.

CHAPTER

FIVE

PARALLELISM API EXAMPLES

In this section, we will introduce how to use EPL *parallelism strategy APIs* to implement different parallelism strategies, as well as their hybrids.

5.1 Data Parallelism

The following snippet shows the data parallelism, where each model replica is placed in one GPU. If the user uses 8 GPUs, then it is a data parallelism task with 8 replicas.

```
import epl
epl.init()
with epl.replicate(device_count=1):
  model()
```

5.2 Pipeline Parallelism

In the following example, the model is divided into two `TaskGraph`s, i.e., "stage_0" and "stage_1". We can set the number of micro batches of the Pipeline by configuring the `pipeline.num_micro_batch` parameter. This model requires two GPUs to place "stage_0" and "stage_1" for each model replica. If the task uses 8 GPUs, EPL will automatically apply a 4-degree data parallelism over the pipeline.

```
import epl

config = epl.Config({"pipeline.num_micro_batch": 4})
epl.init(config)
with epl.replicate(device_count=1, name="stage_0"):
  model_part1()
with epl.replicate(device_count=1, name="stage_1"):
  model_part2()
```

5.3 Tensor Model Parallelism

5.3.1 Large-scale Image Classification

The following example applies different strategies to different parts of the model. We apply data parallelism for the `resnet` part and apply tensor model parallelism to the `classification` part. To reduce the communication overhead among the two taskgraphs, we set `cluster.colocate_split_and_replicate` to colocate the two taskgraphs to the same devices.

```
import epl
config = epl.Config({"cluster.colocate_split_and_replicate": True})
epl.init(config)
with epl.replicate(8):
  resnet()
with epl.split(8):
  classification()
```

5.3.2 MOE Transformer

The following example shows the implementation of a MoE model. We split the tensors for MoE, and set the default strategy as `replicate` for the remaining operations.

```
import epl
config = epl.Config({"cluster.colocate_split_and_replicate": True})
epl.init(config)
total_gpu_num = epl.Env.get().cluster.total_gpu_num

epl.set_default_strategy(epl.replicate(total_gpu_num))

AttentionAndGating()

with epl.split(total_gpu_num):
  MOE_Variable_Define()

MOE_Calculation_Define()
```

CHAPTER

SIX

CONFIGURATION

Users can enable EPL optimized features by configuration.

The configuration tables include:

- Param Key: parameter name, which is defined in the format of "param_category.attribute". `param_category` is the category of parameterse.g., `pipeline`. The `attribute` is the detailed configuration attribute, e.g., `num_micro_batch`.
- Type: parameter type, e.g. str/float/integer/bool
- Default: default value
- Description: parameter description

Configuration APi:

```
Config(param_dict=None)
```

Args	Type	Required	Description
param_dict	dict	False	Parameter dict, where key is the parameter key and value is the parameter value.

Example:

```
import epl
config = epl.Config({"pipeline.num_micro_batch": 4})
epl.init(config)
```

In the above example, we set the configuration by passing a `param_dict`.

You can refer to the following configuration tables for the full parameters.

6.1 Pipeline Configuration

Param Key	Type	Default	Description
"pipeline.num_micro_batch"	integer	1	Pipeline number of micro batches.
"pipeline.num_stages"	integer	None	If `auto.auto_parallel` is True, you can set `pipeline.num_stages` to automatically partition pipeline stages.
"pipeline.strategy"	str	"PreferBackward"	Pipeline schedule policies can be one of ["PreferBackward", "PreferForward"]

- `PreferBackward`: pipeline strategy similar to PipeDream.
- `PreferForward`: pipeline strategy similar to GPipe.

6.2 Gradient Checkpoint (GC) Configuration

Gradient checkpoint reduces the peak memory by saving the activation memory consumption through re-computation.

Param Key	Type	Default	Description
"gradient_checkpoint.type"	str	""	Type to select checkpoint tensor, can be one of ("collection", "auto"). "collection": user selected GC tensors. "auto": automatically searching the GC tensors by analyzing the model.
"gradient_checkpoint.end_taskgraph"	integer	-1	The last taskgraph index to apply GC.
"gradient_checkpoint.check_gradients"	bool	False	Validate the GC gradients.

Examples:

Automatic GC works well for Transformer models.

```
import epl
# Enable auto GC.
config = epl.Config({"gradient_checkpoint.type": "auto"})
epl.init(config)
```

Users can also specify the checkpoint tensors by adding them to a collection, shown as follows:

```
import tensorflow as tf
import epl
config = epl.Config({"gradient_checkpoint.type": "collection"})
epl.init(config)

# specify a checkpoint tensor
tensor = op1()
tf.add_to_collection("checkpoints", tensor)
```

6.3 Zero Configuration

ZeRO leverages the aggregate computation and memory resources of data parallelism to reduce the memory and compute requirements of each device (GPU) used for model training. You can refer to DeepSpeed ZeRO for more information.

Param Key	Type	Default	Description
"zero.level"	str	""	ZeRO levelEPL now supports "v1", which partitions the optimizer states and gradients.

```
import epl

config = epl.Config({"zero.level": "v1"})
epl.init(config)
```

Note

1. EPL ZeRO works only for data parallelism.
2. Now ZeRO cannot be used with gradient accumulation.
3. ZeRO only works for GPU cluster of Nx1 configuration, i.e., N workers, and each worker with one GPU.

6.4 Offload Configuration

EPL supports training large models by offloading weight to CPU memory.

Users can offload parameters by setting `offload.level`.

- "v0": offload all weight to CPU.

Param Key	Type	Default	Description
"offload.level"	str	""	offload level.

Example:

```
import epl
config = epl.Config({"offload.level": "v0"})
epl.init(config)
```

6.5 Memory-efficient AMP Configuration

Memory-efficient AMP does not keep the FP16 weight in memory, instead, EPL casts the weight when needed.

Users can enable EPL AMP by setting `amp.level`.

Param Key	Type	Default	Description
"amp.level"	str	""	Auto mixed precision level, currently only supports O1.
"amp.debug_log"	bool	False	Enable amp debug log.
"amp.loss_scale"	integer/str	"dynamic"	Loss scale for amp, can be "dynamic" or number(for fix).

Example:

```
import epl
config = epl.Config({"amp.level": "O1", "amp.loss_scale": "dynamic"})
# fixed loss scaling
config = epl.Config({"amp.level": "O1", "amp.loss_scale": 128})
epl.init(config)
```

6.6 Optimizer Configuration

Optimizer-related configuration. When updating the parameters in the training process, some user-defined optimizers will consume a large amount of temporary tensor buffers, which increases the peak memory a lot. We can set `num_apply_group` to save memory by updating parameters in groups.

Param Key	Type	Default	Description
"optimizer.num_apply_group"	integer	1	Number of gradient apply groups.

Example:

```
import epl
config = epl.Config({"optimizer.num_apply_group": 30})
epl.init(config)
```

6.7 Cluster Configuration

Param Key	Type	Default	Description
"cluster.device_place_prefer_intra_node"	bool	True	Prefer placing one model replica within node.
"cluster.run_visible_devices"	str	""	Visible devices for session. Usually, its value is set by the scheduler.
"cluster.colocate_split_and_replicate"	bool	False	If cluster.colocate_split_and_replicate is set to Truedifferent taskgraphs will be co-locate in the same device.

6.8 Communication Configuration

Param Key	Type	Default	Description
"communication.num_communicators"	integer	2	number of communicators.
"communication.sparse_as_dense"	bool	False	Whether to transform sparse tensor to dense tensor before communication.
"communication.max_splits"	integer	5	Max number of communication groups for tensor fusion.
"communication.fp16"	bool	False	Enable FP16 AllReduce.
"communication.fp16_scale"	integer	128	Scale the gradients after FP16 AllReduce.
"communication.clip_after_allreduce"	bool	False	Clip gradients after AllReduce.
"communication.gradients_reduce_method"	str	"mean"	AllReduce type, can be one of ("mean", "sum")

6.9 IO Configuration

Param Key	Type	Default	Description
"io.slicing"	bool	False	Whether to slice the dataset.
"io.unbalanced_io_slicing"	bool	False	Allow unbalanced dataset slicing.
"io.drop_last_files"	bool	False	Partition the data files evenly, and drop the last files that cannot be divided.

6.10 Auto Parallel Configuration

Param Key	Type	Default	Description
"auto.auto_parallel"	bool	False	Whether to enable automatic parallelism. (Experimental)

CHAPTER

SEVEN

DATA PARALLELISM

In this section, we will show how to scale the training of ResNet-50 model with EPL data parallelism.

EPL can easily transform the local bert training program to a distributed one by adding a few lines of code.

```
+ import epl
+ epl.init()
+ epl.set_default_strategy(epl.replicate(device_count=1))

ResNet50()
training_session()
```

The following command launches a data parallelism program with two model replicas over two GPUs.

```
epl-launch --num_workers 2 --gpu_per_worker 1 scripts/train_dp.sh
```

`scripts/train_bert_base_dp.sh` is a local training script, `epl-launch` will automatically launch a distributed training program by configuring cluster information.

You can refer to EPL ResNet Example for detailed implementation.

CHAPTER

EIGHT

PIPELINE PARALLELISM

In this section, we will show how to scale the training of Bert model with EPL pipeline parallelism.

8.1 Training setup.

The model code is based on https://github.com/google-research/bert .

8.1.1 Get pretrained bert base model.

```
wget https://storage.googleapis.com/bert_models/2018_10_18/uncased_L-12_H-768_A-12.zip
unzip uncased_L-12_H-768_A-12.zip
```

8.1.2 Prepare dataset

```
mkdir data
cd data
wget https://rajpurkar.github.io/SQuAD-explorer/dataset/train-v1.1.json
wget https://rajpurkar.github.io/SQuAD-explorer/dataset/dev-v1.1.json
wget https://raw.githubusercontent.com/allenai/bi-att-flow/master/squad/evaluate-v1.1.py
```

8.2 Distributed Bert training

8.2.1 Pipeline parallelism

To implement Bert pipeline parallelism, EPL only needs to change the annotation and configuration, as follows:

```
+ import epl
+ epl.init(epl.Config({"pipeline.num_micro_batch": 4}))

# model annotation
+ epl.set_default_strategy(epl.replicate(1))
model_stage0()
+ epl.set_default_strategy(epl.replicate(1))
model_stage1()
```

You can refer to EPL Bert Example for detailed implementation.

The following command launches a pipeline parallelism program with two stages.

```
epl-launch --num_workers 1 --gpu_per_worker 2 scripts/train_bert_base_dp.sh
```

8.3 Evaluation

After training, you can perform the following commands to get the evaluation results.

```
SQUAD_DIR=data
python $SQUAD_DIR/evaluate-v1.1.py $SQUAD_DIR/dev-v1.1.json ${output_dir}/predictions.
↪json
```

You are expected to get f1 ~= 88.0, exact_match ~= 79.8 after 2 epochs.

CHAPTER

NINE

MOE TENSOR MODEL PARALLELISM

This repo contains MoE (Mixture of Experts) transformer training examples with EPL.

9.1 Training setup.

The model code is based on https://github.com/tensorflow/tensor2tensor .

9.1.1 Prepare dataset

Refering to https://github.com/tensorflow/tensor2tensor#adding-a-dataset, script for `translate_ende_wmt32k` shows as following:

```
t2t-datagen --data_dir=data --tmp_dir=data/original/dataset --problem=translate_ende_
↪wmt32k
```

Or, set `FLAGS.generate_data` in `scripts/train_moe_t5.sh` to generate dataset for problem `FLAGS.problem` automatially.

9.2 Distributed Training

To implement MoE tensor model parallelism, EPL only needs to change the annotation and configuration, as follows:

```
+ import epl
+ config = epl.Config({"cluster.colocate_split_and_replicate": True})
+ epl.init(config)
+ epl.set_default_strategy(epl.replicate(total_gpu_num))

AttentionAndGating()

+ with epl.split(total_gpu_num):
    MOE_Variable_Define()

MOE_Calculation_Define()
```

You can refer to EPL MOE Example for detailed implementation.

The following command launches a tensor model parallelism program with two workers.

```
epl-launch --num_workers 2 --gpu_per_worker 1 scripts/train_moe_t5.sh
```