EPL

Alibaba-inc

2023 年 03 月 31 日

EPL

EPL (Easy Parallel Library) 是一个高效易用的分布式大模型训练框架。

CHAPTER 1

概览

Easy Parallel Library (EPL) 是一个高效易用的分布式模型训练框架。EPL 提供了简单易用的 API 来表达各种并行化策略，用户仅需几行代码就可以轻松支持各种模型的高性能分布式训练。

EPL 深度集成了各种训练优化技术，帮助更多的用户低成本，高性能，轻松开启大模型训练。

- 支持各种并行化策略及混合并行，用户仅通过转换并行化接口来实现不同并行化策略训练。
- 支持各种显存优化技术，包含自动 Gradient Checkpoint，ZERO，CPU Offload 技术等，帮助用户用更少的资源训练更大的模型。
- 支持通信优化技术，实现高效的分布式扩展性。

EPL 助力了最大的中文多模态模型 M6 实现大规模分布式训练，通过 512 卡即可训练 10 万亿参数模型。

1.1 使用 EPL 添加分布式策略

通过添加几行代码，用户即可实现不同的并行化策略。完整的 API 介绍和并行化例子详见 API。你也可以参考使用教程来训练 EPL 模型库例子。

数据并行

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

流水并行

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

(续下页)

(接上页)

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

在上述例子中，模型被切分成 2 部分，用户可以通过配置 `pipeline.num_micro_batch` 参数来设定 Pipeline 的 micro batch 数量。

算子拆分

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

在上述例子中，我们对 ResNet 模型部分进行数据并行，对分类层进行算子拆分。

1.2 Citation

```
@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}
}
```

1.3 联系我们

欢迎给我们提 issue, 或者加入 EPL 官方钉钉群。


EPL用户交流群
EPL
扫一扫群二维码，立刻加入该群。

CHAPTER 2

安装

本文档介绍如何搭建 EPL 的运行环境。

2.1 依赖

- TensorFlow-GPU 1.15

2.2 从源码安装

2.2.1 基于 NVIDIA TF1.15 镜像

```
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 基于 TensorFlow TF1.15 镜像

```
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 3

快速开始

我们将通过一个简单的模型示例来演示如何使用 EPL 来实现一个分布式训练程序。

3.1 EPL 分布式策略表达

用户首先需要在本地模型定义文件 `local_model.py` 上添加分布式策略的定义。下面这个例子展示了一个通过添加三行代码实现数据并行的例子。

```
# 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):
```

(续下页)

(接上页)

```
    train_loss, _, step = sess.run([loss, train_op, global_step])
    print("Iteration %s , Loss: %s ." % (step, train_loss))
print("Train Finished.")
```

3.2 启动分布式训练

定义好模型之后，用户需要提供一个本地单级单卡启动的训练脚本，比如 `run.sh`.

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

通过下面的脚本我们可以拉起一个单机两卡的数据并行训练任务。

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

CHAPTER 4

并行化接口

本文档主要介绍了 EPL 并行化原语接口定义，和接口使用的注意事项。

在开始介绍接口定义之前，用户需要了解以下基本概念：

- 模型副本：用户定义的单机单卡模型（不包含任何并行化和 GA 操作）。
- *micro batch size(mb)*: 一个模型副本训练迭代一步学习的 samples 数量。
- *num_micro_batch*: 一个模型副本 GA 或 pipeline 累计的 micro batch 数量。
- *global batch size*: 假设我们对一个模型副本做数据并行操作，并行度为 N，则 global batch size 为 `N * mb * num_micro_batch`.
- *TaskGraph*: TaskGraph 是一个并行化子图。

如果没有特殊说明，EPL 默认用户定义的 batch size 为 `micro batch size`。

4.1 Parallel Strategy 原语

EPL 通过 strategy annotation 的方式来划分模型为多个 `TaskGraph`，并在此基础上进行并行化。EPL 有两类 strategy：`replicate` 和 `split`。每个 strategy 会定义一个 `TaskGraph`。

4.1.1 replicate

`replicate` 可以实现模型的数据并行计算，即将模型复制多份，每份模型副本消费不同数据，来实现计算的并行化。`replicate` scope 下的子模型会构成一个 `TaskGraph`。

1. 当整个模型都标记了 `replicate`，当前只有一个 TaskGraph 做复制，就是传统的数据并行模式。
2. 当部分模型标记了 `replicate`, EPL 会对这部分 TaskGraph 做复制。

接口定义：

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

Args	Required	Description
device_count	True	`replicate` scope 下一个模型副本用来计算的卡数。
name		strategy name

对于数据并行，一个模型副本用一张卡来计算，EPL 会根据当前资源总数推断出全局的副本数。当 `device_count` 大于 1 的时候，EPL 在做模型复制的时候会对 micro batch size 进行拆分，平均到 `device_count` 张卡上，保持用户模型的 micro batch size 保持不变。

示例：

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

上面这个例子是一个数据并行的例子，每个模型副本用一张卡来计算。如果用户申请了 8 张卡，就是一个并行度为 8 的数据并行任务。

4.1.2 split

`split` 可以实现模型的 tensor 拆分计算。`split` scope 下定义的子模型会构成一个 `TaskGraph`，该 `TaskGraph` 会被拆分后放置在多张卡上计算。

接口定义：

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

Args	Required	Description
device_count	True	split 对应的 taskgraph 拆分到 device_count 张卡上计算
name		strategy name

示例：

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

上述例子将模型拆分到 8 张卡上做计算，如果用户申请了 16 张卡，EPL 会默认在拆分模型外面嵌套一层数据并行。

4.2 set_default_strategy

除了两个基本的并行化接口 `replicate` 和 `split` 外，EPL 也提供了一个设置默认 strategy 的辅助接口，如果用户调用 set_default_strategy 方法，会设置一个默认的并行化策略和对应的 `TaskGraph`。这个接口可以帮助模型并行化表达更简洁，同时更灵活地表达出复杂的并行策略。

接口定义：

```
set_default_strategy(strategy)
```

Args	Required	Description
strategy	True	并行化策略。

示例：

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

上述例子设置了一个默认的 `replicate` 策略，通过这种方式也可以实现模型的数据并行。

4.3 接口使用说明与要求

- 不同并行化 strategy 生成的 TaskGraph 默认会放置在不同 Device 上。
- Strategy annotation 不允许嵌套。
- 用户只需标记模型前向代码，backward 和 apply 自动与 Forward colocate 在对应的 `TaskGraph` 里。

关于如何使用并行化接口实现更多灵活的并行化策略，比如 Pipeline，混合并行等，您可以继续阅读并行化例子。

CHAPTER 5

接口使用范例

本文档主要介绍如何使用 EPL 的并行化接口实现常见的并行化策略，以及复杂的混合并行。

5.1 数据并行

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

上面这个例子是一个数据并行的例子，每个模型副本用一张卡来计算。如果用户申请了 8 张卡，就是一个并行度为 8 的数据并行任务。

5.2 流水并行

```
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()
```

在上述例子中，模型被切分成 2 个 `TaskGraph` "stage_0" 和"stage_1"，用户可以通过配置 `pipeline.num_micro_batch` 参数来设定 Pipeline 的 micro batch 数量。在这个例子里，"stage_0" 和"stage_1" 组成一个模型副本，共需要 2 张 GPU 卡。如果用户申请了 8 张卡，EPL 会自动在 pipeline 外嵌套一层并行度为 4 的数据并行（4 个 pipeline 副本并行执行）。

5.3 算子拆分

5.3.1 算子拆分 - 大规模图像分类

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

上述是一个大规模图像分类的例子，在这个例子中，对图像特征部分采用数据并行，对分类层采用算子拆分的方式。为了减少两个 TaskGraph 直接的通信开销，我们可以通过设置 `cluster.colocate_split_and_replicate` 参数将两个 TaskGraph 放置在相同的卡上（默认不同的 TaskGraph 会放置在不同的卡上）。

5.3.2 算子拆分 - MOE Transformer

```
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()
```

在上述例子中，我们实现了一个简单的 MOE 模型，通过设置 `set_default_strategy` 设置默认的并行化策略为 `replicate`, 并对 MOE 部分进行计算的拆分。

CHAPTER 6

配置

用户可以通过配置项开启各种优化功能。目前可以通过环境变量或者配置接口的方式来更改默认配置。

以下配置表格包含

- Param Key: 参数名, 在 EPL 中，参数名的命名规则为"param_category.attribute", `param_category` 为参数的分类，比如 `pipeline`, `attribute` 为每个参数类别下的配置属性，比如 `num_micro_batch`。
- Type: 参数类型
- Default: 默认值
- Description: 解释

接口定义：

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

Args	Type	Required	Description
param_dict	dict	False	参数字典，key 为参数名，value 为参数值。

示例：

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

在上述例子中用户通过构造一个 dict 类型的参数字典，来修改参数配置。具体的参数描述请查阅下文的参数列表。

6.1 Pipeline 配置

Param Key	Type	Default	Description
”pipeline.num_micro_batch”	integer	1	Pipeline number of micro batches.
”pipeline.num_stages”	integer	None	如果开启了自动 Pipeline, 可以配置 pipeline 的 stage 数。
”pipeline.strategy”	str	”PreferBackward”	Pipeline 调度策略，可选策略为 [”PreferBackward”, ”PreferForward”]

- `PreferBackward`: 优先后向计算的调度策略，类似 PipeDream.
- `PreferForward`: 优先前向计算的调度策略，类似 GPipe.

6.2 Gradient Checkpoint (GC) 配置

Gradient checkpoint 通过重算换显存的方式来降低训练过程中的峰值显存。

Param Key	Type	Default	Description
”gradient_checkpoint.type”	str	””	Gradient checkpoint 选点方式，现提供两种方式，”collection”: 用户指定 GC tensor, ”auto”: epl 自动选点
”gradient_checkpoint.end_taskgraph”	integer	-1	当开启 auto GC，用于指定 GC 的结束 taskgraph。
”gradient_checkpoint.check_gradients”	bool	False	校验 GC 生成的梯度的正确性。

代码示例：

自动 GC 选点, 对于 Transformer 类模型，推荐使用 auto GC 的方式。

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

手动选点

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

# 手动指定checkpoint tensor
tensor = op1()
tf.add_to_collection("checkpoints", tensor)
```

6.3 Zero 配置

在数据并行的场景下，每个卡上会存放一个模型副本，optimizer state 等，这些信息在每张卡上都是一样，存在很大的冗余量。当模型变大，很容易超出单卡的显存限制。

在分布式场景下，我们可以通过类似 zero 的思路，将 optimizer state 和 gradient 分片存在不同的卡上，从而减少单卡的 persistent memory 占用。

Param Key	Type	De-fault	Description
”zero.level”	str	””	ZERO 开启级别，目前 EPL 支持 level 设置为”v1”，对 optimizer states 和 gradients 进行拆分。

```
import epl

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

注意：

1. epl zero 只能应用于数据并行部分。
2. 目前不支持 zero 组合 gradient accumulation 使用。
3. 支持的 GPU cluster 为多机一卡的场景，即多个 worker，每个 worker 一张卡。

6.4 Offload 配置

当模型参数量很大，超出 GPU 显存限制，我们可以通过 CPU Offload，利用内存来扩大单卡可以训练的模型规模。epl 可以通过设置 offload.level 来实现 offload。

- ”v0”: offload 所有的参数到 CPU 上。

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

示例

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

6.5 Memory-efficient AMP 配置

TF 原生的 AMP 设计会在显存中保留一份 FP16 的 weight，对于参数量很大的模型，会额外增加显存占用。为了让 AMP 显存开销更友好，EPL 实现了一版 memory-efficient AMP, 通过实时转换和释放的方式来节省显存。

用户可以通过配置 amp.level 参数来开启 EPL 的 AMP。

Param Key	Type	Default	Description
”amp.level”	str	””	Auto mixed precision level, currently only support 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).

示例

```
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 配置

训练任务在做参数更新的时候 (optimizer apply), 对于一些有较多临时 tensor buffer 的 optimizer 实现，容易消耗较多的显存。可以通过配置 num_apply_group 参数实现分组 apply 的方式节省显存消耗。

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

示例

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

6.7 Cluster 配置

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 setted by scheduler.
”cluster.colocate_split_and_replicate”	bool	False	如果 cluster.colocate_split_and_replicate 设为 True，不同的 task-graph 会共享相同的 device。

6.8 通信配置

Param Key	Type	Default	Description
”communication.num_communicators”	integer	2	通信线程池的 communicator 个数。
”communication.sparse_as_dense”	bool	False	是否将 sparse tensor 转换为 dense tensor 进行通信。
”communication.max_splits”	integer	5	最大通信梯度融合的分组数。
”communication.fp16”	bool	False	是否开启 fp16 参数通信。
”communication.fp16_scale”	integer	128	开启 fp16 参数通信后，为防止梯度消失问题，梯度 scale 系数。
”communication.clip_after_allreduce”	bool	False	选择通信后进行梯度 Clip，还是在梯度 Clip 后进行通信。
”communication.gradients_reduce_method”	str	”mean”	梯度 AllReduce 的方式，可以是”mean” 和”sum”。

6.9 IO 配置

Param Key	Type	Default	Description
”io.slicing”	bool	False	是否自动对数据进行分片。
”io.unbalanced_io_slicing”	bool	False	允许数据分片切分时 worker 分配的文件数目不相同，部分 worker 会多分配 1 个训练文件。
”io.drop_last_files”	bool	False	对文件列表进行均匀切分，丢弃多余的文件。

6.10 Auto Parallel 配置

Param Key	Type	Default	Description
”auto.auto_parallel”	bool	False	是否打开自动并行化（目前还在实验阶段）。

CHAPTER 7

Env

本文档主要介绍 EPL 的 Env 中获取常用的运行时信息。

你可以通过 `epl.Env.get()` 获取当前的 env 对象。

7.1 cluster

cluster 包含当前分布式任务的集群信息。

Attribute	Type	Description
`cluster.worker_num`	int	worker 数量
`cluster.worker_index`	int	当前 worker 的 index

示例

```
import epl
env = epl.Env.get()
worker_num = env.cluster.worker_num
worker_index = env.cluster.worker_index
```

7.2 config

config 包含当前 epl 的配置信息。

示例

```
import epl
env = epl.Env.get()
config = env.config
```

CHAPTER 8

数据并行

本节将介绍如何通过 EPL 来对 `ResNet-50` 模型做数据并行分布式训练。

通过添加以下几行代码，EPL 即可将本地训练程序转换成分布式训练程序。

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

ResNet50()
training_session()
```

用户可以通过以下脚本来启动一个 2 卡的数据并行训练任务。

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

`scripts/train_dp.sh` 是一个本地的训练脚本。

完整的训练代码可以参考EPL ResNet Example。

CHAPTER 9

流水并行

本节将介绍如何通过 EPL 来对 Bert 模型做 Pipeline 分布式训练。

9.1 训练准备

这个例子采用的 Bert 模型代码基于 Google 官方的 Bert Repo https://github.com/google-research/bert 。

9.1.1 下载预训练 Bert 模型文件

```
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
```

9.1.2 准备数据集

```
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
```

9.2 分布式 Bert 流水并行训练

用户仅需要添加几行并行化策略和配置代码，即可实现 Bert 的流水并行训练策略。

```
+ 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()
```

用户可以通过以下脚本来启动一个 2 个 stage 的流水并行训练任务。

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

完整的训练代码可以参考 EPL Bert Example。

9.3 模型验证

完成训练后，你可以通过以下脚本来得到验证结果。

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

在模型训练 2 Epoch 后，预期会得到 f1 ~= 88.0, exact_match ~= 79.8。

CHAPTER 10

MoE 算子拆分并行

本节将介绍如何通过 EPL 来实现 MoE (Mixture of Experts) transformer 模型训练。

10.1 训练准备

模型代码将基于tensor2tensor的组件。

10.1.1 准备数据集

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

或者，通过在 `scripts/train_moe_t5.sh` 脚本中设置 `FLAGS.generate_data` 来自动下载和准备数据。

详细的准备流程可以参考 tensor2tensor 文档.

10.2 分布式训练

EPL 仅需添加几行代码来实现 MoE 算子拆分并行，如下所示：

```
+ 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):
```

(续下页)

(接上页)

```
  MOE_Variable_Define()

MOE_Calculation_Define()
```

用户可以通过以下脚本来启动一个 2 卡的 MOE 算子拆分并行训练任务。

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

完整的训练代码可以参考 EPL MOE Example。