LLMのABCI上での学習

Shukai Nakamura

@idaten459

自己紹介

- 中村秋海
- 普段は富岳とかABCIで大規模並列学習をおこなってます
- 最近は通信の高速化について少し取り組んでます

導入

- 大規模言語モデル(LLM)は最近盛り上がってる
  - chatGPT, Bard, ...
  - スケーリングのためには分散学習が必要
- 今日からあなたもABCI上で分散学習できるようになる！！
- 要点だけ話すので，詳細はスライドを見返して欲しい

説明すること・しないこと

- 説明すること
  - ABCI上での環境構築
  - Megatron-DeepSpeedの使い方
  - ABCI上での分散学習
    - 複数ノードでの実行
- 説明しないこと
  - ABCI以外での計算機クラスタの使い方
    - 応用はできると思う
  - 言語モデルの基礎知識
    - NLPの知識は僕より深い人が多そう

環境構築

前提

- ABCIアカウントを持っていて，ssh 経由でアクセスできる
- V100(16GB)ノードを用いる
- 作業ディレクトリは `$WORK_DIR` とする
- グループ名は `$GROUP` とする
- 作業レポジトリ
  - https://github.com/rioyokotalab/Megatron-DeepSpeed-Ylab

仮想環境の作成

- python仮想環境はvenvを用いて行う
- https://docs.abci.ai/ja/python/ のPython仮想環境参照.

実行コード例

```
module load python/3.10/3.10.10                          # load python 3.10
python3 -m venv $WORK_DIR/megatron-deepspeed-abci        # create virtual environment
source $WORK_DIR/megatron-deepspeed-abci/bin/activate    # activate virtual environment
```

Megatron-DeepSpeedの環境構築

- ABCI用のスクリプトを同梱した `https://github.com/rioyokotalab/Megatron-DeepSpeed-Ylab.git` を用いる
- オリジナルの `https://github.com/microsoft/Megatron-DeepSpeed.git` を用いても問題ないように書く予定
  - `megatron/arguments.py` のみ書き換えが必要

```
cd $WORK_DIR
git clone https://github.com/rioyokotalab/Megatron-DeepSpeed-Ylab.git
cd Megatron-DeepSpeed/dataset
sh download_books.sh # データセットのダウンロード(学習したいものを用意する)
sh download_vocab.sh # vocabファイルのダウンロード
```

pythonの環境構築

- 仮想環境を作成したら，必要なライブラリをインストールする.
- PyTorchはCUDA 11.8に対応している `torch==2.0.0+cu118` をインストールする.
  - requirements.txtを2-5行目を以下のように書き換える
  - Ylabバージョンは書き換えてある

```
pybind11
# for install pytorch version cu118
--find-links https://download.pytorch.org/whl/torch_stable.html
torch==2.0.0+cu118
#torchaudio==2.0.1+cu118
torchvision==0.15.1+cu118
six
regex
numpy
```

pythonの環境構築2

ライブラリのインストール

```
cd $WORK_DIR
cd Megatron-DeepSpeed-Ylab
pip install -r requirements.txt
```

DeepSpeedのインストール

- DeepSpeedをインストールする

```
pip install deepspeed
```

CUDAなどのライブラリのロード

```
module load cuda/11.8/11.8.0 cudnn/8.6/8.6.0 nccl/2.16/2.16.2-1
module list # ロードされているか確認
# > Currently Loaded Modulefiles:
# 1) python/3.10/3.10.10   2) cuda/11.8/11.8.0   3) cudnn/8.6/8.6.0   4) nccl/2.16/2.16.2-1
```

apexのインストール

- ソースコードからビルドする.
- (多分)GPUやCUDAに依存してビルドするのでインタラクティブジョブに入ってビルドする.
- ABCIのジョブの投げ方は後で触れる

ログインノード

```
qrsh -g $GROUP -l rt_G.small=1 -l h_rt=1:00:00
```

計算ノード

```
cd $WORK_DIR
source megatron-deepspeed-abci/bin/activate
module load python/3.10/3.10.10 cuda/11.8/11.8.0 cudnn/8.6/8.6.0 nccl/2.16/2.16.2-1
git clone https://github.com/NVIDIA/apex
cd apex
pip install -v --disable-pip-version-check --no-cache-dir \
 --global-option="--cpp_ext" --global-option="--cuda_ext" ./
```

ABCIのジョブ実行方法

以下の3つのジョブが存在する

- インタラクティブジョブ
  - コマンドがその場で帰ってくるジョブ
  - デバッグ用に使われる
- バッチジョブ
  - コマンドが書かれたシェルスクリプトを実行するジョブ
  - スケジューラによってスケジュールされ実行される
  - 学習は一般にこのジョブで行う
- 事前予約ジョブ
  - 特定の時間に特定のノードを計画的に実行されるジョブ
  - 時間単価が高い

ABCIの資源タイプ

- Vノード
  - V100 16GBを搭載したノード
  - 1ノードあたり最大4GPU
  - 最大512ノード
- Aノード
  - A100 40GBを搭載したノード
  - 1ノードあたり最大8GPU
  - 最大64ノード
- 詳しくは https://docs.abci.ai/ja/job-execution/ 参照

ABCIのジョブの投げ方

インタラクティブジョブ

```
qrsh -g グループ名 -l リソースタイプ=ノード数 [オプション]
```

バッチジョブ

```
qsub -g グループ名 [オプション] submit.sh
```

シングルノードの実行

- GPT2(345M)の学習例
  - バッチジョブでの実行
  - 時間をかなり短めに設定しているため，学習は終わらない
- `GROUP` に自分の所属しているグループを入れる
- 詳しくはgithub上のsubmit_pretrain_gpt_abci.sh参照

```
# 1ノード1GPU, 40 TFLOPSくらい(V100の理論ピークは125 TFLOPS)
qsub -g $GROUP submit_pretrain_gpt_abci.sh
```

```
# 1ノード4GPU, DP=4, 40 TFLOPS/gpuくらい
qsub -g $GROUP submit_pretrain_gpt_distributed_abci.sh
```

複数ノードの実行

- 複数ノードにまたがる場合 `mpirun` を用いる
- ノード間通信のために `MASTER_ADDR` と `MASTER_PORT` を設定する
- 通信ノードを指定する `hostfile` を設定する

```
# マスターアドレス（複数ノード実行の際に、親となるノードのIPアドレス）取得
export MASTER_ADDR=$(/usr/sbin/ip a show dev bond0 | grep inet | cut -d " " -f 6 | cut -d "/" -f 1)
# マスターポート（ジョブごとにユニークなポート番号）
export MASTER_PORT=$((10000 + ($JOB_ID % 50000)))
# ホストファイル（ABCIは環境変数 SGE_JOB_HOSTLIST で取得できる）
HOSTFILE_NAME=./hostfile_${JOB_ID}
cat $SGE_JOB_HOSTLIST > $HOSTFILE_NAME
```

複数ノードの例

```
# mpirunの実行例
mpirun -np $WORLD_SIZE -npernode $GPUS_PER_NODE --hostfile $HOSTFILE_NAME python example.py
```

- 詳しくはgithubレポジトリ参照

```
# 8ノード 4GPU/nodeのジョブの例
qsub -g $GROUP submit_pretrain_gpt_distributed_multinodes_with_mp_abci.sh
```

Megatron-DeepSpeedとMPIの組み合わせの注意点

- Megatron-DeepSpeedのランクやワールドサイズの環境変数の想定
  - `RANK`, `WORLD_SIZE`
- MPIの動作
  - `OMPI_COMM_WORLD_RANK`, `OMPI_COMM_WORLD_SIZE`
- 書き換えが必要

3種類の並列

- データパラレル（DP）
  - 各プロセスごとにモデルを配置して，各プロセスで計算した勾配をプロセス間で同期してパラメータを更新する手法
  - 1GPU 1プロセスにすることで，だいたいGPU数倍速くなる
    - (ミニバッチサイズをそろえた場合)
- モデルパラレル（MP）
  - テンサーパラレルともいう
- パイプラインパラレル（PP）
- DP，MP，PPそれぞれを組み合わせて使うこともできる

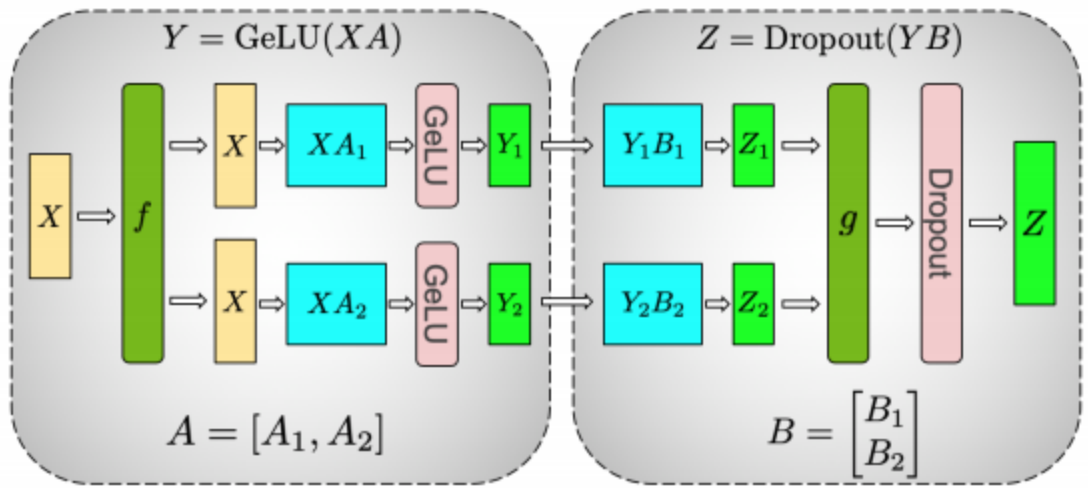
データ並列

- 各プロセスごとにモデルを配置して，各プロセスで計算した勾配をプロセス間で同期してパラメータを更新する手法
- 1GPU 1プロセスにすることで，だいたいGPU数倍速くなる
  - (ミニバッチサイズをそろえた場合)
- PyTorchはDistributedDataParallelを参照
- Megatron-DeepSpeedの場合WorldSize(並列数)からMPサイズとPPサイズを割った値が自動的にDPになる
  - 割り切れない時はassertion error?

$$\mathrm{DP\_SIZE} = \frac{\mathrm{WORLD\_SIZE}}{\mathrm{MP\_SIZE} \times \mathrm{PP\_SIZE}}$$

モデルパラレル

- 水平方向にモデルを分割
- 層内でGPUに分割することで，1プロセスが所持するモデルサイズが小さくなり，CPUやGPUのメモリに乗り切らないサイズの大規模なモデルも扱うことできる
- 1GPU当たりの計算の量が少なくなるのでモデル全体として早くなる



(a) MLP

[1]より引用

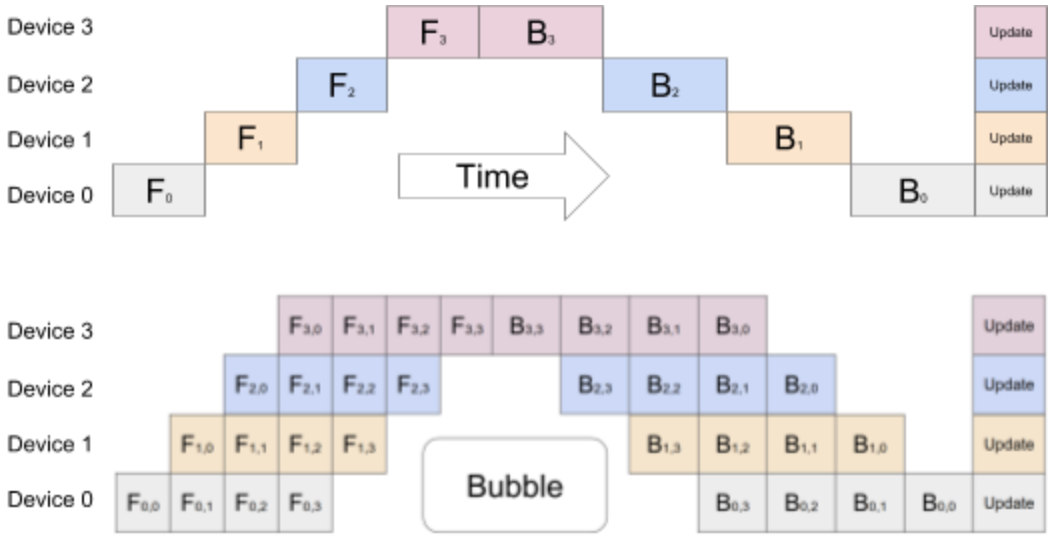
モデルパラレルの使い方

- PyTorchでは，distributed tensor parallelを参照
- Megatron-DeepSpeedでは`pretrain_gpt.py`などの実行時引数に以下を加える
  - 対応していないモデルもある

```
# 例，実際には(url追加)参照
python pretarin_gpt.py
  --tensor-model-parallel-size $TENSOR_MODEL_PARALLEL_SIZE
```

パイプラインパラレル

- 垂直方向にモデルを分割
- 層毎にGPUに分割することで，1プロセスが所持するモデルサイズが小さくなる
- バッチを分割しパイプライン化することで，モデル分割のオーバーヘッドを減らしている



[2]より引用

パイプラインパラレルの使い方

- 現状PyTorchでは既存のモデルを`torch.nn.Sequential()`に書きかえる必要がある
  - https://pytorch.org/docs/stable/pipeline.html 参照
- Megatron-DeepSpeedなら`pretrain_gpt.py`などの実行時引数に以下を加える

```
# 例，実際には(url追加)参照
python pretarin_gpt.py \
  --pipeline-model-parallel-size $PIPELINE_MODEL_PARALLEL_SIZE
```

大規模言語モデルの学習

- メモリが足りない時
  - ミニバッチサイズを小さくする
  - MP，PPサイズを大きくする
  - Activation Checkpointingなどの手法を使う
- 13BのモデルをV100で学習する場合
  - mbs=1, mp=4, pp=8, ac=trueで学習できるようになる

参考文献

[1] Mohammad Shoeybi et al. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv.

[2] Yanping Huang et al. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. NeurIPS.