

パイプライン並列分散深層学習の一実装手法の提案

滝澤 尚輝¹ 矢崎 俊志² 石畑 宏明¹

概要: 本論文では、並列計算機におけるパイプライン並列深層学習の一実装手法を提案し、その評価・分析を行う。パイプライン並列ではニューラルネットワークモデルを分割し、各プロセスに割り当てる。ハードウェア効率を向上させるため、ミニバッチを分割したマイクロバッチを用いて各プロセスの処理をオーバーラップする。パイプライン並列の利点はマイクロバッチ処理のオーバーラップによる高速化と、メモリ消費の分散である。本研究では、パイプライン並列におけるニューラルネットワークモデルの分割の記述方法を提案する。全結合層 32 層からなるシンプルなネットワークを用いてパイプライン並列の高速化の効果について分析を行う。

Proposing an Implementation Method for Pipeline Parallelism Distributed Deep Learning

1. はじめに

近年、ニューラルネットワークの大規模化に伴って深層学習の精度が向上している。これはハードウェアの性能の向上やメモリ容量の増加による。しかし、ハードウェアの性能向上以上にニューラルネットワークの大規模化が進んでいる。

ニューラルネットワークの大規模化に伴う学習時間の増加やメモリ消費量の増加という問題を解決するために分散深層学習が用いられている。分散深層学習では主にデータ並列手法が用いられている。効率的な速度向上が期待でき、かつ実装が容易なためである。しかし、データ並列ではニューラルネットワーク全体をメモリに配置する必要がある。そのため、メモリ容量以上の大規模なニューラルネットワークを単純なデータ並列で学習することはできない。

データ並列と異なる高速化手法としてパイプライン並列の研究が行われている。パイプライン並列ではメモリ消費の分散も期待できるため、メモリにのりきらない大規模なニューラルネットワークの学習にも適用することができる。

本研究では並列計算機におけるパイプライン並列分散深層学習の実装手法について提案する。計算時間や通信時間からパイプライン並列の効果について分析する。

2. 関連研究

Chiceon Kim らはマルチ GPU 環境で利用できる GPipe[1] に基づくパイプライン並列分散深層学習のライブラリを実装した [2]。GPipe と同様に、順伝播で計算した結果を一度削除し、逆伝播時に再計算することでメモリ消費量を削減している。また、CUDA ストリームを用いることで通信のオーバーラップを行っている。AmoebaNet-D と U-Net を用いて評価を行っている。AmoebaNet-D では、2 並列のモデル並列と比較して、最大で 4.95 倍の高速化を達成した。U-Net では、並列化しない場合と比べて 3.105 倍の高速化を達成した。

Deepak Narayanan らは PipeDream というパイプライン並列ライブラリの提案を行った [3]。PipeDream では単一 GPU を用いて各層の計算時間などを計測し、モデルの分割を動的に最適化している。計算量の多い層ではデータ並列と組み合わせることで速度を向上させている。順伝播と逆伝播を交互に実行することで待機時間を削減し効率を向上させている。様々なニューラルネットワークやハードウェア構成で比較を行った結果、PipeDream は最大で 5.3 倍の高速化を達成した。

3. 分散深層学習

3.1 データ並列

データ並列では学習データを各プロセスに分散する手法

¹ 東京工科大学
Tokyo University of Technology

² 電気通信大学
The University of Electro-Communication

である。各プロセスは割り当てられた学習データに対して順伝播、逆伝播を行い、得られた各層のパラメータの勾配を AllReduce 通信により平均化する。並列数の増加に伴って効率よく高速化することが期待できる。ニューラルネットワーク全体をメモリに配置する必要があるため、メモリサイズを超える大規模なニューラルネットワークにはそのまま適用することができない。

3.2 モデル並列

モデル並列ではニューラルネットワークを分割し各プロセスに割り当てる手法である。順伝播、逆伝播それぞれにおいて、各プロセスは前のプロセスからデータを受け取り、次のプロセスへ計算結果を送信する。ニューラルネットワークが分割されることからメモリ消費量が削減される。したがって、メモリサイズを超えるニューラルネットワークを学習する時に適用される。

3.3 パイプライン並列

パイプライン並列ではモデル並列と同様にニューラルネットワークを分割し、各プロセスに割り当てる。ミニバッチを分割しパイプライン処理を行うことで、計算をオーバーラップさせる。パイプライン並列の処理を図1に示す。図1では4並列でのパイプライン並列を表している。入力データが4つに分割され矢印で示したようにデータが流れる。青い矢印で示した部分は待機時間を表している。

4. パイプライン並列の実装

本研究では chainer[4] を用いてパイプライン並列を実装した。

4.1 ニューラルネットワークの分割

パイプライン並列におけるニューラルネットワークの分割を容易にする手法として、chainer.Sequential クラスを拡張し Divisible クラスを実装した。

パイプライン並列ではニューラルネットワークの分割が必要である。一般的に用いられている chainer.Chain クラスを用いて分割する場合、ソースコード1のように、分割後のネットワークモデルをそれぞれ chainer.Chain クラスを用いて実装する必要がある。この方法ではニューラルネットワークや並列数などが変更された場合に実装しなおす必要があり、スケラビリティが低いと言える。

chainer.Sequential クラスはレイヤを配列として保持している。python の配列では array[start:end] とすることで配列から目的の要素のみを取り出すことができる。したがって、chainer.Sequential クラスを拡張し、Divisible クラスを実装した。Divisible クラスによる分割はソースコード2のように行う。この実装により、分割後の開始レイヤと最終レイヤのインデックスを指定するだけで分割が可能

となる。

4.2 パイプライン並列の実装

実装したパイプライン並列深層学習における、1ミニバッチを処理するプログラムをプログラム3に示す。comm.rank とは分散処理において、各プロセスに割り当てられる ID である。comm.size とはプロセス数である。分割されたニューラルネットワークは comm.rank が 0 のプロセスから順に割り当てられ、comm.rank が 0 のプロセスが入力層、comm.rank が comm.size-1 のプロセスが出力層を担当する。

順伝播では、初めに comm.rank が 0 のプロセスはミニバッチからマイクロバッチの生成、それ以外は comm.rank-1 のプロセスからデータを受信する。生成、または受信したデータを用いて順伝播を実行し、comm.rank が comm.size-1 のプロセスは誤差を求める。逆伝播に必要であるため、入力と出力を保持する。最後に comm.rank が comm.size-1 でないプロセスは comm.rank+1 のプロセスへ順伝播の計算結果のデータを送信する。これらの処理をミニバッチ全体を処理し終えるまで繰り返す。

逆伝播では、初めに順伝播で保持した出力から1つ取り出す。comm.rank が comm.size-1 以外のプロセスは comm.rank+1 のプロセスから出力の勾配を受信する。逆伝播を実行する。comm.rank が 0 のプロセス以外は comm.rank-1 のプロセスへ入力の勾配を送信する。保持しているすべての出力に対して逆伝播を実行するまでこれらの処理を繰り返す。

4.3 通信

通信は chainermn を利用している。chainermn の通信関数では、初めに送受信する配列の変数型や要素数の通信を行う。受信する側は受信した変数型や要素数からバッファを用意する。最後に配列の通信を行う。

順伝播では、comm.rank-1 のプロセスから計算結果の配列を受信する。受信した配列から chainer.Variable クラスのインスタンスを生成する。

逆伝播では、comm.rank+1 のプロセスから勾配の配列を受信する。受信した勾配配列を対応する Variable インスタンスの勾配配列に設定する。

5. パイプライン並列動作確認

パイプライン並列が正しく動作していることを確認するために実験を行った。ニューラルネットワークは 3072 入力 3072 出力の全結合層を 3 層、3072 入力 10 出力の全結合層を 1 層、計 4 層とした。データセットは Cifar-10 を用いた。パイプライン並列と並列化を行わないシリアルな処理で誤差の比較を行った。

実験結果を図2に示す。横軸はエポック、縦軸は誤差を

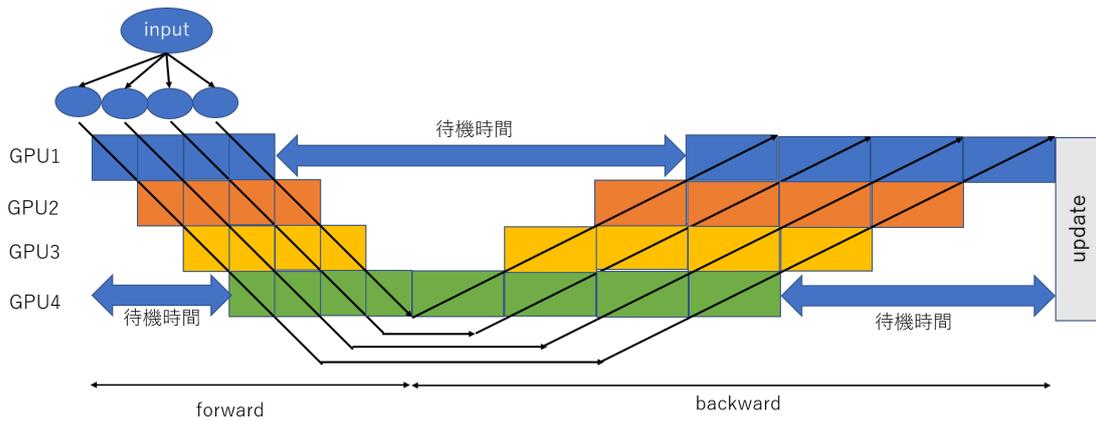


図 1 パイプライン並列の処理の概要

プログラム 1 chainer.Chain クラスによる分割

```

1 class NN0(chainer.Chain):
2     # rank:0 用のモデル
3     def __init__(self):
4         self.layer = ...
5     def forward(self, x):
6         x = self.layer(x)
7         return x
8
9 class NN1(chainer.Chain):
10    # rank:1 用のモデル
11    def __init__(self):
12        self.layer = ...
13    def forward(self, x):
14        x = self.layer(x)
15        return x
16
17 comm = chainermn.create_communicator('nccl')
18 if comm.rank == 0:
19     model = NN0()
20 elif comm.rank == 1:
21     model = NN1() # rank により生成するクラスが違う

```

プログラム 2 Divisible クラスによる分割

```

1 class NN(Divisible):
2     def __init__(self):
3         self.append(Layer_class())
4         ...
5
6 model = NN()
7 start = 分割後開始レイヤのインデックス
8 end = 分割後最終レイヤのインデックス
9 model = model.divide(start, end)

```

プログラム 3 パイプライン並列の実装

```

1 # forward
2 for mb in range(0, batchsize, mbatchsize):
3     if comm.rank == 0:
4         mx = x[mb:mb+mbatchsize]
5         if comm.rank == comm.size-1:
6             mt = t[mb:mb+mbatchsize]
7         if comm.rank != 0:
8             mx = recv_data(comm, None, source)
9         xs.append(mx)
10        y = model(mx)
11        if comm.rank == comm.size-1:
12            loss = F.softmax_cross_entropy(y, mt)
13            acc = F.accuracy(y, mt)
14            ys.append(loss)
15        else:
16            send_data(comm, y, dest)
17            ys.append(y)
18
19 # backward
20 for x, y in zip(xs, ys):
21     if comm.rank != comm.size-1:
22         y = recv_grad(comm, y, dest)
23     y.backward()
24     if comm.rank != 0:
25         send_grad(comm, x, source)

```

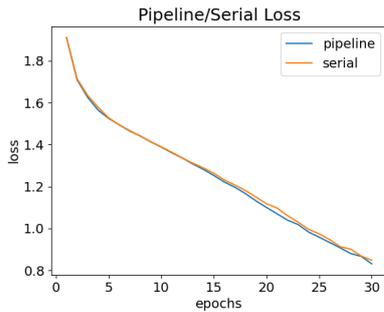


図 2 パイプライン並列/シリアル 誤差比較

表 1 reedbush-1 構成

プロセッサ名	Intel Xeon E5-2695v4 (Broadwell-EP)
プロセッサ数 (コア数)	2 (36)
メモリ容量	256 GB
メモリ帯域幅	153.6 GB/sec
GPU	NVIDIA Tesla P100 x4
GPU メモリ容量	16 GB
GPU メモリ帯域幅	732 GB/sec
CPU-GPU 間接続	PCI Express Gen3 x16 レーン (16 GB/sec)
GPU 間通信	NVLink 2 brick(20 GB/sec x1 or 2)
インターコネクト	InfiniBand EDR 4x2 リンク (100 Gbps x2)

表している。図 2 から、誤差の変化が凡そ同じであり、パイプライン並列は正しく動作していると言える。

次に NVidia Visual Profiler(NVVP) を用いてパイプライン並列の動作を確認した。ニューラルネットワークは 8192 入力, 8192 出力の全結合層 31 層, 8192 入力, 10 出力の全結合層 1 層から構成されるネットワークを用いた。データセットはサイズが 8192 の 0 埋め配列を用い, ミニバッチサイズは 32, ミニバッチの分割数は 4 とした。実験環境は東京大学情報基盤センターの reedbush-1 を利用した。2 ノード, 2 プロセス/ノードの計 4 プロセスで実行した。reedbush-1 の構成を表 1 に示す。

NVVP の表示を図 3 に示す。図 3 において, 赤は順伝播, 黄色は逆伝播, 緑はパラメータ更新, 青は通信および待機時間を表している。青い矢印はデータの流れを表している。プロセス 0 と 1 および 2 と 3 は同一ノード内のプロセスであるため, 0 と 1, 2 と 3 はノード内 NVLink による通信, 1 と 2 はノード間 InfiniBand による通信が行われている。図 1 と図 3 を比較すると, 逆伝播の計算時間が 1 回目とそれ以降で異なることがわかる。これはパラメータの勾配の加算が行われていることが原因であると考えられる。2 回目以降の逆伝播では 1 回目の逆伝播で計算されたニューラルネットワークのパラメータの勾配が存在するため, 既に存在する勾配と新しく計算した勾配の加算が行われる。

6. 実験

32 層の全結合層で構成されるニューラルネットワークを用いて実際に学習を行い, ミニバッチ 1 回あたりの学習時間を計測した。ニューラルネットワークの出力数は 10, ミニバッチサイズは 1200 とした。学習時間の計測では, 10 回のミニバッチの学習時間の平均を計算した。

6.1 ミニバッチ分割数

ミニバッチの分割数による学習時間を図 4 に示す。データサイズは 5000, 1 ノードあたり 1 プロセスとした。並列化しない場合の実行時間に対する高速化率を図 5 に示す。

1 プロセスでは計算のオーバーラップが行えないため, ミニバッチを分割すると遅くなる。2 プロセス, 4 プロセスではミニバッチの分割数が 5 の時に最も高速で, それぞれ並列化しない場合の 1.34 倍, 1.93 倍であった。8 プロセス, 16 プロセス, 32 プロセスではミニバッチの分割数が 20 の時に最も高速で, それぞれ 2.7 倍, 3.49 倍, 3.72 倍であった。ミニバッチを分割しない場合, プロセス数が増えるほど通信回数が増えることから速度が遅くなる。プロセス数が増えるほど, 最も高速となるミニバッチ分割数が大きくなる。最も高速となるミニバッチ分割数よりもミニバッチを細かく分割すると速度が遅くなり, 2 プロセスでは 12 以上の時にミニバッチを分割しない時より低速となる。本実験では最大のミニバッチ分割数を 80 としたが, これ以上分割するとより低速になると考えられる。

6.2 通信帯域幅

通信帯域幅の違いによる学習時間を図 6 に示す。4 ノードで各ノード 1 プロセスの実行において, 通信はすべて InfiniBand で行われる。1 ノード 4 プロセスの実行において, 通信は NVLink で行われる。図 6 から, 通信帯域幅による違いはほとんどないと考えられる。

6.3 入力データサイズ

入力データサイズによる高速化率の比較を行った。各プロセスにおいてミニバッチを分割せずに実行する場合の実行時間と比較した高速化率を用いた。4 プロセスで比較した場合の図 7 に示す。8 プロセスで比較した場合の図 8 に示す。16 プロセスで比較した場合の図 9 に示す。図 7, 8, 9 すべてにおいて, 入力データサイズが 3000 より小さくなると高速化率が小さくなる。図 8, 9 において, 入力データサイズが 3000 以上ではマイクロバッチサイズ 120 が最も高速となり, 入力データサイズが 2000 以下では 2 番目に低速となる。4 プロセス, 8 プロセスでは入力データサイズ 10000 でほとんどのマイクロバッチサイズの高速化率が小さくなるが, 16 プロセスではほとんど変化がない。

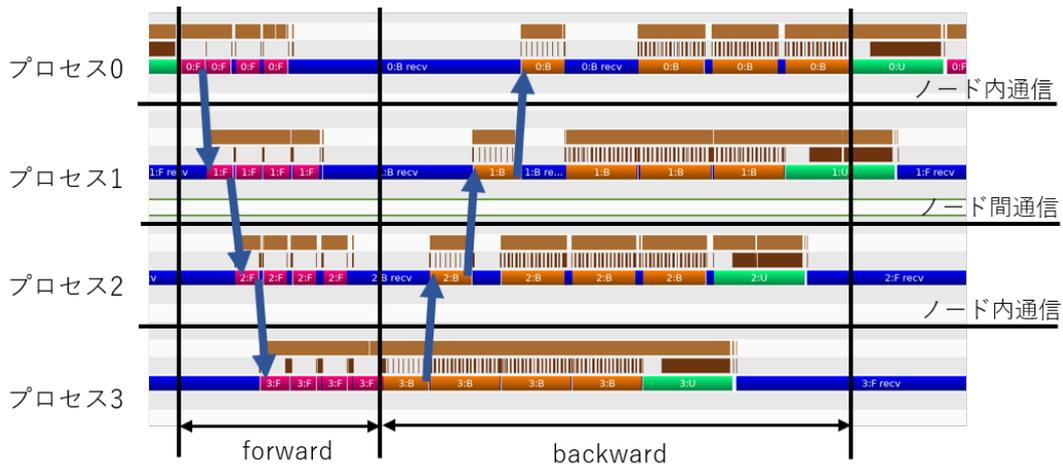


図 3 NVidia Visual Profiler での動作確認結果

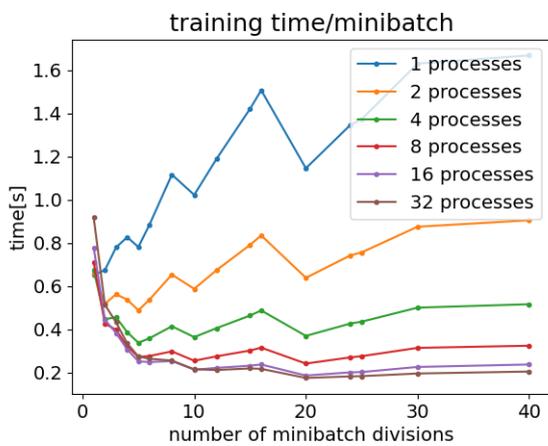


図 4 ミニバッチ分割数による 1 ミニバッチの学習時間
1 ノードあたり 1 プロセス, データサイズは 5000, ミニバッチサイズは 1200

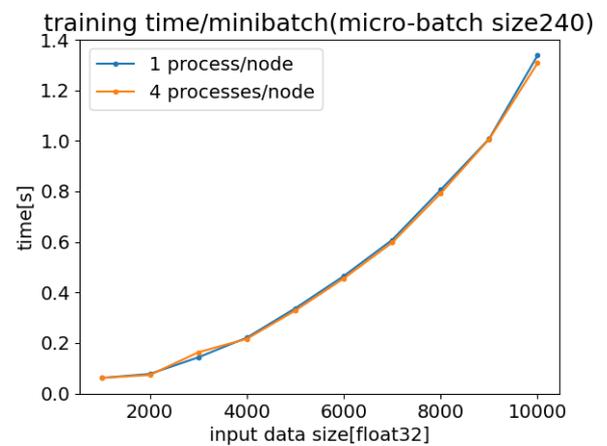


図 6 通信帯域幅の違いによる 1 ミニバッチの学習時間

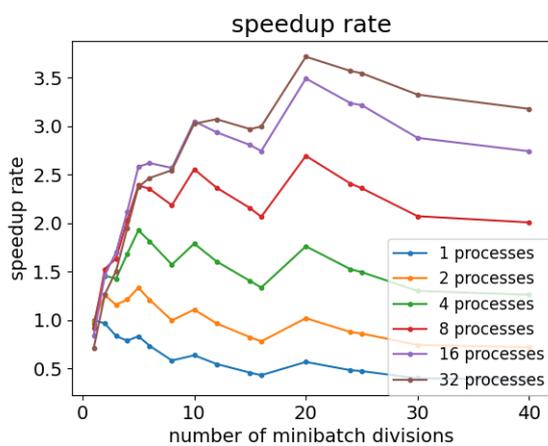


図 5 ミニバッチ分割数による高速化率
条件は図 4 と同様, 1 プロセスでミニバッチを分割しない場合の
実行時間を基準とした高速化率

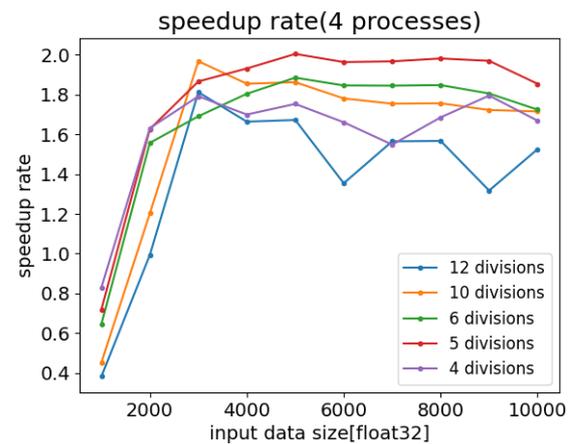


図 7 4 プロセスにおける入力データサイズによる高速化率
同プロセス数でミニバッチを分割しない場合を基準とした高速化率

7. 考察

1 回当たりの通信時間 t_{comm} は式 1 で求められる.

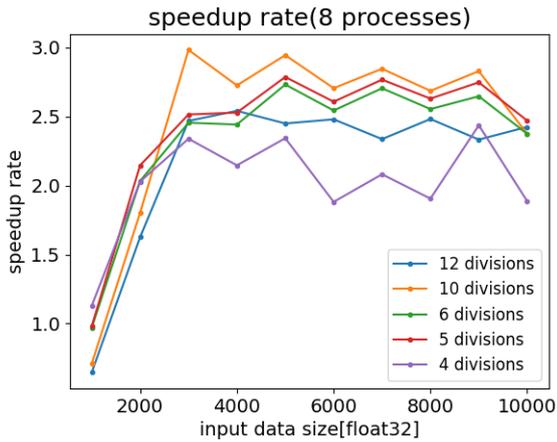


図 8 8 プロセスにおける入力データサイズによる高速化率

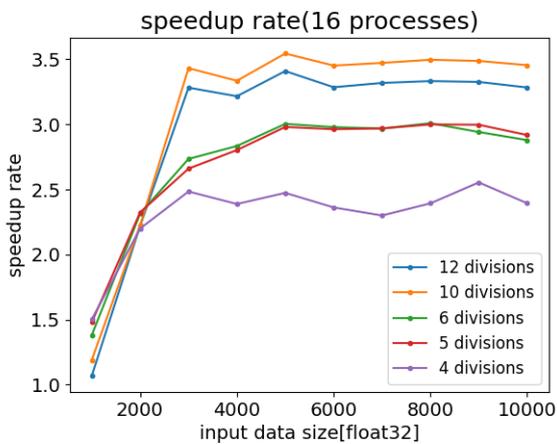


図 9 16 プロセスにおける入力データサイズによる高速化率

$$t_{comm}(bs) = t_0 + \frac{bs \times dsiz e}{bw} \quad (1)$$

bs はバッチサイズ, t_0 は通信の立ち上がり, $dsiz e$ は通信されるデータサイズ, bw は通信帯域幅を表す.

並列化しない場合の計算時間を t_{comp} とすると, パイプライン並列での計算時間 t_{pipe} は式 2 のように表される.

$$t_{pipe} = (m + d - 1) \times \frac{1}{m} \times \frac{t_{comp}}{d} \quad (2)$$

m はミニバッチの分割数, d はニューラルネットワークの分割数を表している. ニューラルネットワークを d 分割しているため, ミニバッチ 1 回の計算時間は $\frac{t_{comp}}{d}$ となる. ミニバッチを m 分割しているため, 分割後のマイクロバッチ 1 回の計算時間は $\frac{1}{m}$ となる. パイプライン処理により計算をオーバーラップするため, マイクロバッチ 1 回の処理は $m + d - 1$ 回となる. 式 2 からわかるように m, d が大きいほど計算時間をオーバーラップすることができるため, 計算時間が短くなる.

パイプライン並列における計算と通信を含めた処理時間 T_{pipe} は式 3 で表される.

$$T_{pipe} = t_{pipe} + (m + d - 2)t_{comm}\left(\frac{bs}{m}\right) \quad (3)$$

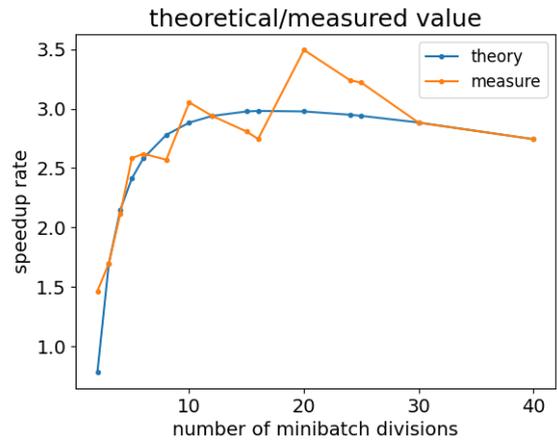


図 10 16 プロセスパイプライン並列における理論値と実測値の比較
理論値はミニバッチ分割数が 3, 12, 40 の時の実測値を用いて近似

ミニバッチの分割数 m が増えても総通信量は変わらないが, 通信回数が $m + d - 2$ 回に増える. d や m の増加に伴って t_{pipe} は減少するのに対し, 通信回数が増えるため $t_0 \times (m + d - 2)$ だけ増加する.

式 3 と図 5 で用いた実測値を用いて理論値と実測値の比較を行った. 結果を図 10 に示す. 計算時間 t_{comp} と通信の立ち上がり t_0 , 通信時間 $\frac{bs \times dsiz e}{bw}$ は, ミニバッチ分割数 3, 12, 40 の時の実測値を用いて求めた. 図 10 から式 3 は十分に近似できていると考えられる.

以上の理由から, 計算時間に対してミニバッチの分割数 m が大きすぎると通信のオーバーヘッドにより速度が遅くなる. 逆に m が小さすぎると計算のオーバーラップが足りず, 高速化の効果が十分に得られない.

入力データサイズが小さいと計算量と通信量も少なくなる. したがって, 計算のオーバーラップによる高速化の効果が小さくなる. 特に図 7, 8, 9 において, 入力データサイズが 1000 の時にマイクロバッチサイズが小さいほうが高速化率が小さくなるのは, 計算時間に対して通信回数が多いためである. 計算のオーバーラップによる高速化の効果に対して通信の立ち上がり時間 t_0 が大きいことが原因と考えられる.

8. まとめ

本研究の目的はパイプライン並列の分析を行い, 有用性を示すことである. chainer を用いてパイプライン並列を実装した. プロセス数やミニバッチ分割数, 入力データサイズなどの条件によるパイプライン並列の効果について考察した.

プロセス数によって最適なミニバッチの分割数が異なることがわかった. ミニバッチの分割数が多すぎると並列化しない場合より遅くなってしまいう場合もある. また, 計算量が小さいと高速化の効果が少なくなることがわかった.

これは計算時間に対して通信の立ち上がり時間が大きいことが原因であると考えられる。

chainer の通信関数では最初に送受信するデータの情報を通信する。1回のデータの通信に2回の通信が行われるため、レイテンシの影響が大きくなる。レイテンシが小さくなるように通信関数を実装することで、より高速化することが期待できる。

参考文献

- [1] Yanping Huang 他, "GPipe: Efficient Training of Giant Neural Network using Pipeline Parallelism", In Advances in Neural Information Processing Systems, pages 103–112, 2019.
- [2] Chiheon Kim 他, "torchpipe: On-the-fly Pipeline Parallelism for Training Giant Models", arXiv:2004.09910, 2020
- [3] Deepak Narayanan 他. "PipeDream: generalized pipeline parallelism for DNN training", SOSP '19: Proceedings of the 27th ACM Symposium on Operating Systems Principles October 2019 Pages 1–15
- [4] 徳井誠也 他. "Chainer: A Deep Learning Framework for Accelerating the Research Cycle", KDD '19: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, July 2019, Pages 2002–2011