情報処理学会論文誌 Vol.63 No.5 1206–1215 (May 2022)

パイプライン並列分散深層学習の一実装手法の評価

滝澤 尚輝[1,a)] 矢崎 俊志[2] 石畑 宏明[1]

受付日 2021年8月22日, 採録日 2022年2月4日

概要：本論文では，並列計算機におけるパイプライン並列分散深層学習の一実装手法の評価・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．VGG16 と ResNet50 を用いて，複雑なモデルにおけるパイプライン並列の評価を行う．

キーワード：分散深層学習，並列処理

Evaluation of an Implementation Method for Pipeline Parallelism Distributed Deep Learning

Naoki Takisawa[1,a)] Syunji Yazaki[2] Hiroaki Ishihata[1]

Received: August 22, 2021, Accepted: February 4, 2022

Abstract: In this paper, we evaluate and analyze an implementation method of pipeline parallelism distributed deep learning on parallel computers. In pipeline parallelism, a neural network model is partitioned and assigned to each process. To improve hardware efficiency, we use microbatches, which are divided minibatches, to overlap the processing of each process. The advantage of pipeline parallelism is that the overlapping of microbatch processes increases the speed and distributes the memory consumtion. In this study, we propose a method for describing the partitioning of neural network models in pipeline parallelism. We analyze the speedup effect of pipeline parallelism using a simple network with 32 fully connected layers. Using VGG16 and ResNet50, we evaluate the pipeline parallelism.

Keywords: distributed deep learning, parallel processing

1. はじめに

近年，深層学習に用いられるニューラルネットワークが大規模化している．大規模なニューラルネットワークは，より複雑な問題を精度良く解くことができる可能性がある．

大規模なニューラルネットワークは計算量とメモリ消費量が多くなる．ニューラルネットワークの大規模化にともな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．データ並列手法は，効率的な速度向上が期待でき，かつ実装が容易である．しかし，データ並列手法ではニューラルネットワーク全体を単一ノードのメモリに配置する必要がある．そのため，単一ノードに実装されたメモリ容量以上の大規模なニューラルネットワークにデータ並列手法を適用することはできない．

小容量のメモリを用いて大規模なニューラルネットワークの学習を行いたい場合もある．理化学研究所と富士通が2021 年に開発したスーパコンピュータ「富岳」は，A64FX と呼ばれる 52 コアの CPU と 32 GiB のメモリを搭載して

[1] 東京工科大学
Tokyo University of Technology, Hachioji, Tokyo 192–0982, Japan
[2] 電気通信大学
The University of Electro-Communications, Chofu, Tokyo 182–8585, Japan
[a)] g212004023@edu.teu.ac.jp

おり，GPU は搭載していない．このように，小規模なノードを多数接続した並列計算環境も存在する．

データ並列手法と異なる高速化手法としてパイプライン並列手法がある [1], [2], [3], [4], [5]．パイプライン並列手法では，メモリ消費を複数ノードに分散することができるため，単一ノードのメモリに載せきれ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．

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 torchgpipe を実装した [4]．torchgpipe では，GPipe と同様に，順伝播で計算した結果を 1 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 [6]．また，torchgpipe では，CUDA ストリームを用いることで通信のオーバラップも行っている．torchgpipe の評価として AmoebaNet-D [7] および U-Net [8] のパイプライン並列を実装し，性能を比較している．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と報告されている．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と報告されている．しかし，torchgpipe はマルチ GPU 環境で利用することを前提としており，並列数は接続されている GPU 数に依存する．

Narayanan らは，パイプライン並列ライブラリ PipeDream を提案した [2]．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．さらに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ことが報告されている．PipeDream では，順伝播と逆伝播を非同期的に実行するため，複数のバージョンのパラメータを保持する必要があり，メモリ消費量の分散が難しい．

Tanaka らは，パイプライン並列とデータ並列を組み合わせたハイブリッド並列分散深層学習を行うミドルウェア RaNNC を提案した [9]．RaNNC では，ニューラルネットワークモデルの分割と分割後の各ステージのデータ並列数を自動で決定する．ニューラルネットワークモデルの分割はデバイスのメモリに載り，かつ，計算時間が可能な限り均等になるように行われる．RaNNC は，パラメータの収束性を劣化させないために，GPipe のような同期型パイプライン並列を採用している．様々な隠れ層や層の総数を用いた BERT モデル [10] と ResNet モデル [11] を BiT [12] と同様に拡張した ResNet152x8 モデルを用いた実験を行った．BERT モデルを用いた実験において，Megatron-LM [13] と比較して 5 倍のモデルの学習に成功し，Megatron-LM と同等のスループットを達成した．BERT モデルおよび ResNet152x8 モデルを用いたいずれの実験においても，試みたすべての条件において GPipe よりも優れたスループットを達成した．

パイプライン並列の処理時間は，分割したニューラルネットワークのうち，最も時間を要する部分の処理時間で律速される．分散深層学習におけるパイプライン並列では，分割後の各ニューラルネットワークの処理時間が等しくなるような分割をすることで，高速化の効果が大きくなる．また，単一ノードに載せきれない大規模モデルの学習に用いられるため，分割においては，メモリ消費量も考慮する必要がある．したがって，パイプライン並列の実装段階において様々な分割戦略を試行錯誤する必要があり，分割位置を容易に変更できる実装であることが重要である．

パイプライン並列を実装した研究は増えているが，データ並列と組み合わせた研究が多く，それらの多くはパイプライン並列とデータ並列を区別せずに評価している．ノード数が限られている場合において，データ並列数を最優先するよりもニューラルネットワークモデルの分割数を増やしパイプラインの最適化を行ったほうがいい場合もある．本研究ではパイプライン並列に絞り，詳細な分析・評価を行う．

本研究では，表 1 の構成で示したような，GPU が搭載されている InfiniBand クラスタ程度のバランスのシステムを対象として，パイプライン並列について分析・評価する．パイプライン並列を含む分散深層学習は，通信性能と演算性能のバランスによって全体の性能が大きく変化するため，本論文の適用範囲は前述のシステム程度のバランスとする．

2. 分散深層学習

2.1 データ並列

データ並列はニューラルネットワークは分割せず，学習データを複数のプロセスに分散する手法である．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．並列数の増加にともなった効率の良い高速化が期待できる [14], [15], [16]．一方，ニューラルネットワークは分割しないため，その全体を各ノードのメモリに配置する必要がある．単一ノードの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．

2.2 モデル並列

モデル並列ではニューラルネットワーク自体を分割し，分割したニューラルネットワークの一部を各プロセスに割り当てる手法である．各プロセスは，順伝播，逆伝播それぞれにおいて，前の処理を担当するプロセスからデータを受け取り，次の処理を担当する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，プ

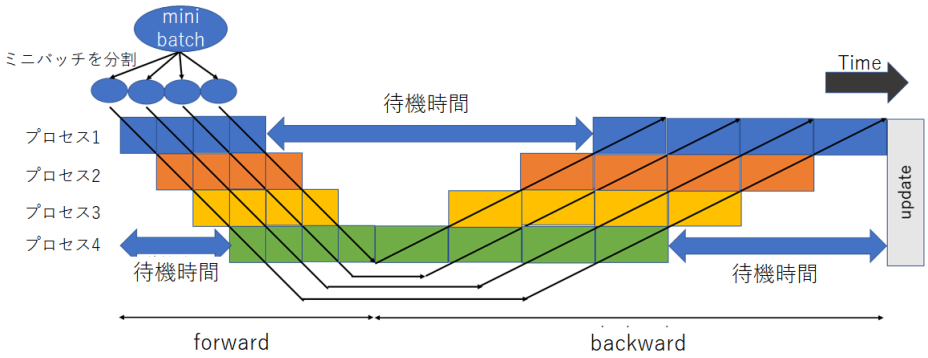

図 1 4 プロセスによるパイプライン並列処理の概要．ミニバッチを 4 つに分割し，黒矢印の流れで処理する．青矢印は，パイプライン処理で生じる待機時間を表す

Fig. 1 Overview of pipeline parallel processing with four processes. The minibatch is divided into four parts and processed by the black arrows. The blue arrows represent the waiting time in the pipeline processing.

ロセスを複数ノードに分散配置することで，メモリの少ないノードの集合でニューラルネットワークを処理することができる．したがって，単一ノードのメモリサイズを超えるニューラルネットワークの処理に適用される．

2.3 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，分割したニューラルネットワークの一部を各プロセスに割り当てる．ニューラルネットワークの計算は，通常，ミニバッチ単位で行うが，ミニバッチを分割しパイプライン処理することで，計算をオーバラップさせる．パイプライン並列の処理を**図 1** に示す．図 1 は 4 つのプロセスによるパイプライン並列処理を表している．図では，入力データ（ミニバッチ）を 4 つに分割している．分割されたデータは，黒矢印で示した流れで処理される．図中の青矢印で示した部分は待機時間を表している．

3. パイプライン並列の実装

本研究では Chainer [17] を用いてパイプライン並列を実装した．

3.1 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 に実装されている `chainer.Sequential` クラスを拡張し，新たに `Divisible` クラスを実装した．

Chainer 標準の `chainer.Chain` クラスを用いて，パイプライン並列のためのニューラルネットワーク分割を行う場合を考える．この場合，**図 2** に示すように，分割後のニューラルネットワーク各部をそれぞれ `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，並列数，分割位置などを変更する場合，すべての分割部分を `chainer.Chain` で実装しなおす必要

```
class NN0(chainer.Chain):
  # rank:0 用のモデル
  def __init__(self):
    self.layer = ...
  def forward(self, x):
    x = self.layer(x)
    return x

class NN1(chainer.Chain):
  # rank:1 用のモデル
  def __init__(self):
    self.layer = ...
  def forward(self, x):
    x = self.layer(x)
    return x

comm = chainermn.create_communicator('nccl')
if comm.rank == 0:
  model = NN1()
elif comm.rank == 1:
  model = NN1() # rank により生成するクラスが違う
```

図 2 chainer.Chain クラスによるニューラルネットワークの分割．各レイヤ（NN0，NN1）を chainer.Chain クラスで実装するため，分割方法を変更する場合は，コード全体の書き直しが必要になる

Fig. 2 To divide the neural network by chainer.Chain class. Each layer (NN0, NN1) is implemented by chainer.Chain class. If you want to change the division method, you need to rewrite the whole code.

があり，実装の柔軟性が低い．

本研究では次のような実装方法を採用する．Chainer の `chainer.Sequential` クラスはニューラルネットワークのレイヤを配列として保持している．Python では，`array[start:end]` と記述することで，配列 `array` から，`start`〜`end` をインデックスとする目的の要素のみを取り出すことができる．これを利用して `chianer.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**図 3** のように行う．この実装により，

```
class NN(Divisible):
  def __init__(self):
    self.append(Layer_class())
    ...

model = NN()
start = 分割後開始レイヤのインデックス
end = 分割後最終レイヤのインデックス
model = model.divide(start, end)
```

図 3 本研究で実装した Divisible クラスによる分割．start と end で分割位置の配列インデックスを指定することで，分割位置を柔軟に変更できる

Fig. 3 The Divisible class implemented in this reserch can flexibly change the division position by specifying the array index of the division position in start and end.

```
# forward
for mb in range(0, batchsize, mbatchsize):
  if comm.rank == 0:
    mx = x[mb:mb+mbatchsize]
  if comm.rank == comm.size-1:
    mt = t[mb:mb+mbatchsize]
  if comm.rank != 0:
    mx = recv_data(comm, None, source)
  xs.append(mx)
  y = model(mx)
  if comm.rank == comm.size-1:
    loss = F.softmax_cross_entropy(y, mt)
    acc = F.accuracy(y, mt)
    ys.append(loss)
  else:
    send_data(comm, y, dest)
    ys.append(y)

# backward
for x, y in zip(xs, ys):
  if comm.rank != comm.size-1:
    y = recv_grad(comm, y, dest)
  y.backward()
  if comm.rank != 0:
    send_grad(comm, x, source)
```

図 4 パイプライン並列の実装

Fig. 4 Implementation of pipeline parallelism.

分割後の開始レイヤと最終レイヤのインデックス（`start` および `end`）を指定するだけで分割位置の変更が可能となる．

3.2 パイプライン並列の実装

本研究で実装した Divisible クラスにより実装したパイプライン並列深層学習において，ミニバッチを 1 つ処理するプログラムを**図 4** に示す．`comm.rank` は分散処理において，各プロセスに割り当てられる ID（MPI Rank）を示す．`comm.size` とは並列処理に参加するプロセス総数を示す．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る．`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を分割する．分割したものをマイクロバッチと呼ぶ．それ以外のプロセスは，それぞれ `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する．`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，すべてのプロセスは自身への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を送信する．この一連の処理を，ミニバッチ全体の処理が終わるまで繰り返す．

逆伝播では，初めにすべてのプロセスが，順伝播の処理中に保持した出力から 1 つを取り出す．`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．その後，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．

3.3 通信

通信は ChainerMN [18] 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を通信相手に通知する．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．バッファの準備が完了したら，配列の内容を送信する．

順伝播では，各プロセスが `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，各プロセスが `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 `chainer.Variable` インスタンスの勾配配列に設定する．

4. パイプライン並列動作確認

本研究で採用するパイプライン並列実装方式が正しく動作することを確認するため，簡易なニューラルネットワークでパイプライン並列の動作確認を行った．ニューラルネットワークは 3,072 入力 3,072 出力の全結合層を 3 層に 3,072 入力 10 出力の全結合層を 1 層加えた計 4 層とした．データセットは CIFAR-10 [19] を用いた．パイプライン並列による処理と，並列化を行わないシリアル処理で訓練誤差を比較した．誤差の変化はほとんど同じで学習は同等に行われていることを確認した．

次に NVidia Visual Profiler（NVVP）を用いてパイプライン並列処理の動作を確認した．

ニューラルネットワークは 8,192 入力，8,192 出力の全結合層 31 層，8,192 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

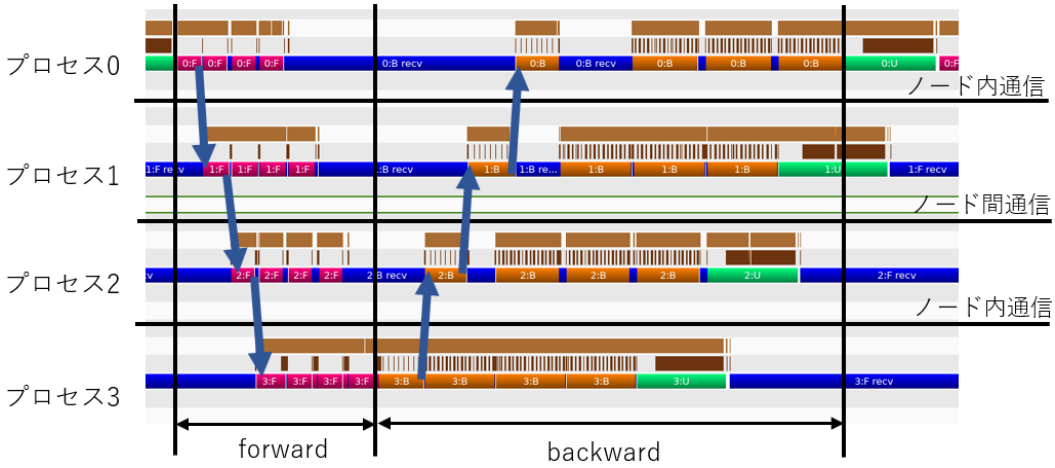

図 5 NVidia Visual Profiler によるプロファイリング結果
Fig. 5 Profiling results by NVidia Visual Profiler.

表 1 Reedbush-L の構成
Table 1 Configuration of Reedbush-L.

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）

8,192 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．処理は，2 ノード，2 プロセス/ノードの計 4 プロセスで実行した．

実験環境は東京大学情報基盤センターの Reedbush-L を利用した．Reedbush-L の構成を**表 1** に示す．

NVVP によるプロファイリング結果を**図 5** に示す．図 5 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．よって，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 5 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分かる．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，すで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．

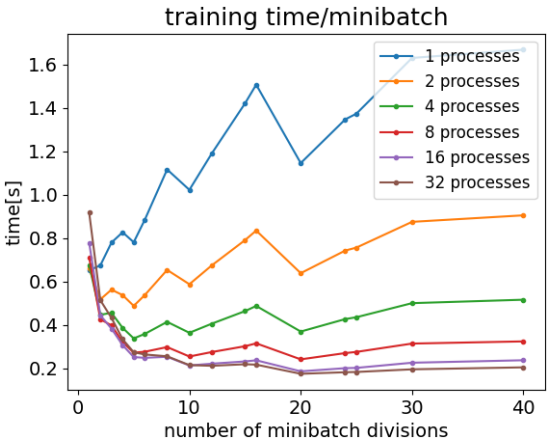

図 6 ミニバッチ分割数による 1 ミニバッチの学習時間．1 ノードあたり 1 プロセス，データサイズは 5,000，ミニバッチサイズは 1,200
Fig. 6 Learning time for one minibatch by number of minibatch divisions. 1 process per node, data size is 5,000, minibatch size is 1,200.

5. 単純なモデルを用いた実験

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学習させる処理において，ミニバッチ 1 回あたりの学習時間を計測した．計測は 10 回行い，その平均を測定値とした．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 1,200 とした．

5.1 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 6** に示す．データサイズを 5,000 とし，1 ノードあたり 1 プロセスで実行した．並列化しない場合の実行時間に対する高速化率を**図 7** に示す．1 プロセスでは計算にオーバラップは発生しないため，ミニバッチを分割すると処理時間は遅くなる．2 プロセスおよび 4 プロセスではミニバッチ分割数が 5 のとき

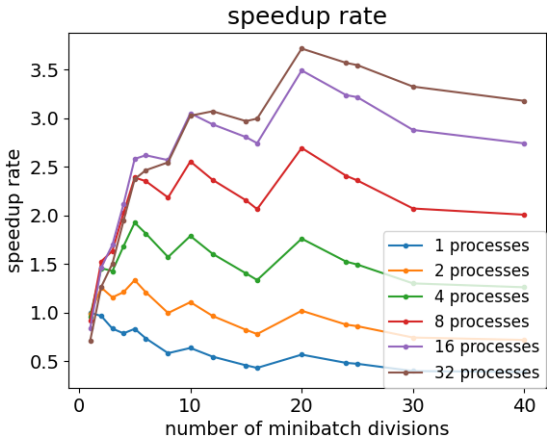

図 7 ミニバッチ分割数による高速化率．条件は図 6 と同様，1 プロセスでミニバッチを分割しない場合の実行時間を基準とした高速化率

Fig. 7 Speed-up ratio by number of minibatch divisions. The condition is the same as Fig. 6, but the speed-up rate is based on the execution time when the minibatch is not divide in one process.

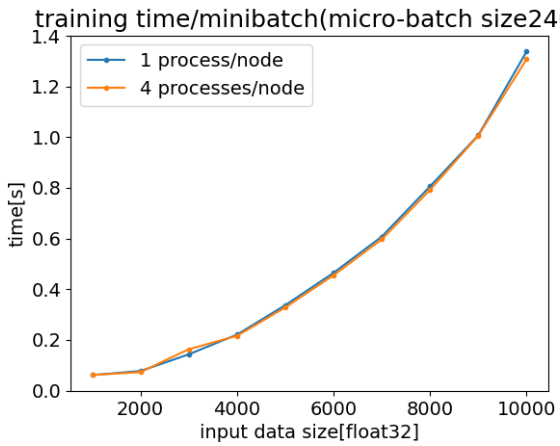

図 8 通信帯域幅の違いによる 1 ミニバッチの学習時間の比較

Fig. 8 Learning time comparison between different communication bandwidths.

に最速で，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，および 32 プロセスではミニバッチ分割数が 20 のときに最速で，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は遅くなる．プロセス数が増えるほど，最速となるミニバッチ分割数が大きくなった．最速となるミニバッチ分割数よりも分割数を増やすと速度が遅くなり，2 プロセスでは分割数 12 以上のときにミニバッチを分割しないときより低速となる．本実験ではミニバッチ分割数の最大を 80 としたが，これ以上分割するとより低速になると考える．

5.2 通信帯域幅

通信帯域幅の違いによる学習時間を**図 8** に示す．1 ノードあたり 1 プロセスの処理を 4 ノードで実行した場合，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 8 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．

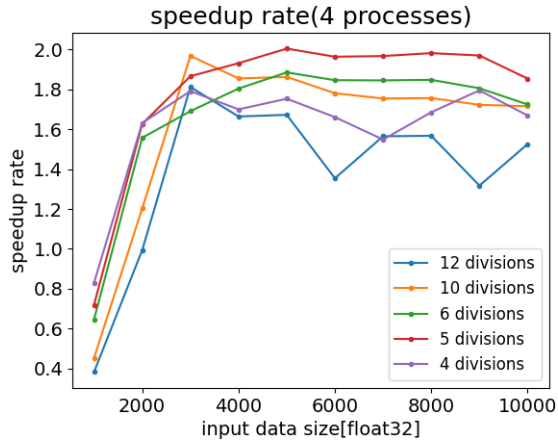

図 9 4 プロセスにおける入力データサイズによる高速化率．同プロセス数でミニバッチを分割しない場合を基準とした高速化率

Fig. 9 Speed-up ratio of 4 processes by input data size. Speed-up rate based on the same number of processes and no minibatch division.

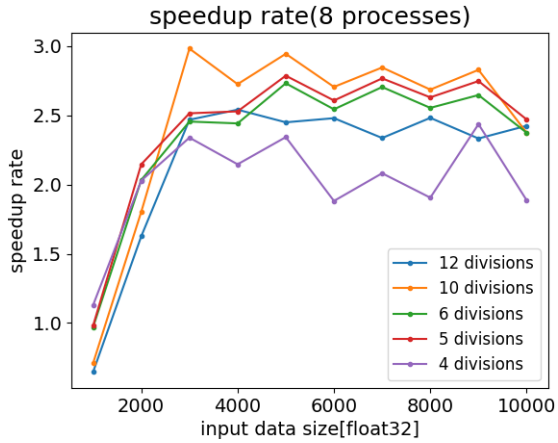

図 10 8 プロセスにおける入力データサイズによる高速化率

Fig. 10 Speed-up ratio with 8 processes.

5.3 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結果を**図 9** に示す．8 プロセスで比較した場合の結果を**図 10** に示す．16 プロセスで比較した場合の結果を**図 11** に示す．図 9，図 10，図 11 すべてにおいて，入力データサイズが 3,000 より小さくなると高速化率は低くなる．図 10 と図 11 において，入力データサイズが 3,000 以上ではマイクロバッチサイズ 120 が最速となり，入力データサイズが 2,000 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 10,000 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

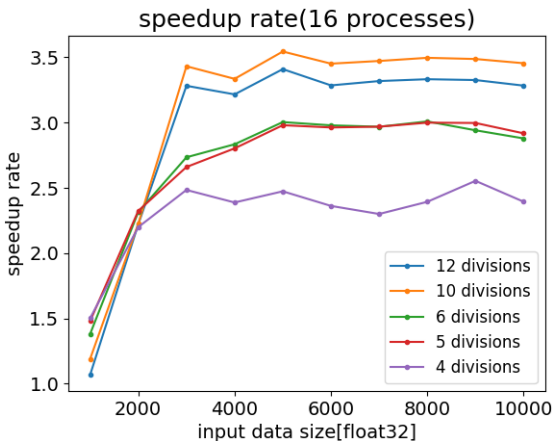

図 11 16 プロセスにおける入力データサイズによる高速化率
Fig. 11 Speed-up ratio with 16 processes.

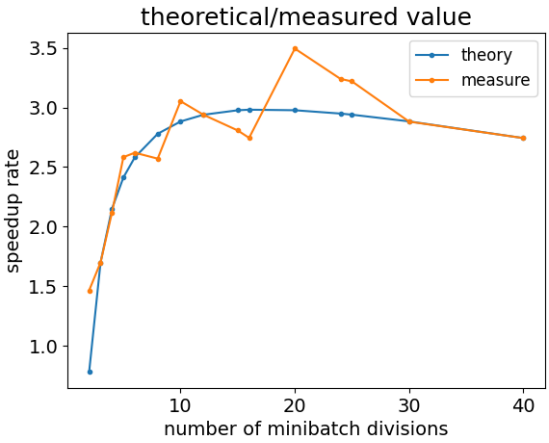

図 12 16 プロセスパイプライン並列における理論値と実測値の比較．理論値はミニバッチ分割数が 3，12，40 のときの実測値を用いて近似
Fig. 12 Comparison of theoretical and measured values for a 16 process pipeline parallelism. Theoretical values are approximated using the measured values when the number of minibatch divisions is 3, 12, and 40.

ロセスではほとんど変化がない．

6. 考察

1 回あたりの通信時間 t_{comm} は式 (1) で求められる．

$$t_{comm}(bs) = t_0 + \frac{bs \times dsize}{bw} \tag{1}$$

bs はバッチサイズ，t_0 は通信の立ち上がり，$dsize$ 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．

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

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) から分かるように m，d が大きいほど計算時間をオーバ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．

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

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，通信回数は $m+d-2$ 回に増える．d や m の増加にともなって t_{pipe} は減少するのに対し，通信時間は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 7 から，理論値と実測値の比較を行った．結果を**図 12** に示す．計算時間 t_{comp} と通信の立ち上がり t_0，通信時間 $\frac{bs \times dsize}{bw}$ は，ミニバッチ分割数 3，12，40 のときの実測値を用いて求めた．図 12 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，計算のオーバラップによる高速化の効果も小さくなる．特に図 9，図 10，図 11 において，入力データサイズが 1,000 のときに，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．

7. VGG16，ResNet50 を用いた実験

VGG16 と ResNet50 を用いて学習を行い，ミニバッチ 1 回あたりの学習時間を計測した．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．

各モデルの分割位置は各プロセスが担当する層数ができる限り等しくなるように分割した．また，ResNet50 は BottleNeck アーキテクチャを最小単位として分割している．BottleNeck アーキテクチャ内では skip connection が使われている．skip connection は，層が深くなることによる勾配消失問題を改善する手法として用いられており，ある層の出力に手前の層の入力を加算する処理のことである．したがって，BottleNeck 内で分割を行うと，より多くの通信が必要となる．VGG16，ResNet50 におけるミニバッチサイズはそれぞれ 84，90 とした．

VGG16 における高速化率を**図 13** に示す．縦軸は並列

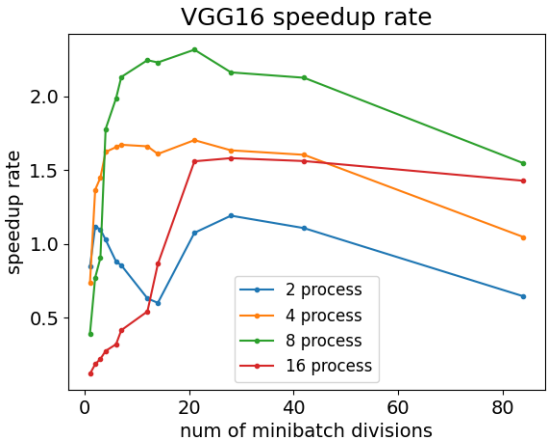

図 13 VGG16 における高速化率
Fig. 13 Speed-up rate in VGG16.

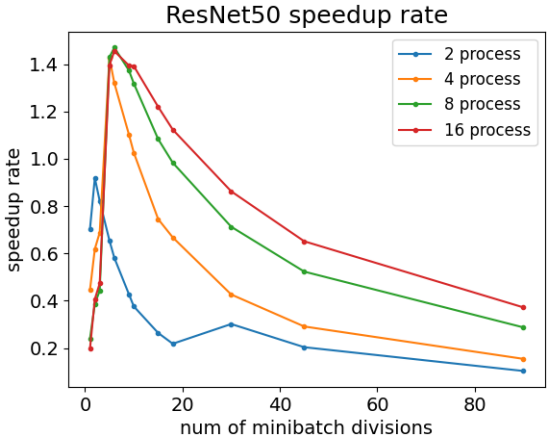

図 14 ResNet50 における高速化率
Fig. 14 Speed-up rate in ResNet50.

化しない場合に対する高速化率，横軸はミニバッチの分割数を表している．

4 プロセス以上で最大 1.5 倍以上の高速化率となった．8 プロセス，ミニバッチ分割数 20 のときに 2.32 倍の高速化率であった．16 プロセスでの高速化率が 8 プロセスや 4 プロセスでの高速化率より低い原因として，分割位置があげられる．本実験では ReLU 関数や max pooling 層などのパラメータを持たない層も分割の対象としているため，モデルを 16 分割したときにこれらの層しか割り当てられないプロセスや全結合層や畳み込み層などのような計算量の多い層が 2 層割り当てられるプロセスがあった．したがって，プロセスによって計算量に偏りがあったことが原因であると考えられる．

ResNet50 における高速化率を図 **14** に示す．

4 プロセス以上では高速化率に大きな差は見られなかった．各プロセスの計算量の偏りが原因であると考えられる．ResNet50 は同じ大きさの BottleNeck アーキテクチャを重ねることで層を深くしているため，VGG16 と比較して，入力層に近い層を割り当てられたプロセスと出力層に近い層を割り当てられたプロセスで計算量に大きな差が生じる．

8. まとめ

本研究ではパイプライン並列において，ニューラルネットワークを簡単に分割できる記述方法を提案，実装した．通信時間と演算時間をモデル化し，通信時間のパイプライン並列分散深層学習への影響を分析した．VGG16 と ResNet50 を用いて実際に利用されているモデルにおけるパイプライン並列の評価を行い，高速化の効果を確認した．特定のプロセスに演算量が偏るとパイプライン並列による高速化の効果が減少することが分かった．

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分かった．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る場合もある．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分かった．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．

VGG16，ResNet50 ともに分割後のニューラルネットワークの計算量が偏ることによって，高速化の効果が小さくなることが確認できた．分割後のニューラルネットワークの計算量ができる限り等しくなるようにモデルを分割することで，パイプラインの各段の処理時間が等しくなるため，高速化率が高くなると考えられる．

本研究では，Chainer を用いてパイプライン並列を実装した．しかし，Chainer は開発が終了し，PyTorch への移行が進められている．本研究では，ChainerMN の通信関数を用いているため，そのまま PyTorch へ移植することはできないが，send や recv などの基本的な通信関数のみを利用しているため，移植は難しくないと考えている．また，Chainer，PyTorch ともに define-by-run であることなど，実装手法が似ていることから，移植した際の性能への影響は小さいと考えられる．

パイプライン並列を利用することにより，単一ノードのメモリ容量を超える大規模なモデルの学習を高速化することが可能になる．本研究では，GPU を搭載している InfiniBand クラスタ程度のシステムを対象としている．GPU と InfiniBand で構成されているスーパコンピュータが多いため，既存のシステムに対し，そのまま適用することができる．単一ノードのメモリ容量を超える大規模モデルを学習できることで，既存システムにおいて，機械学習のパフォーマンスの向上が期待できる．

今後は様々な分割手法を実装し，評価・分析を行う．また，図 7 から分かるように，必要以上にパイプライン並列数を増やしても効果が少ない．したがって，ニューラルネットワークモデルは単一ノードのメモリに載る最低限の分割を行い，パイプライン並列とデータ並列を組み合わせる方法について検討する．

参考文献

[1] Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q.V., Wu, Y., et al.: Gpipe: Efficient training of giant neural networks using pipeline parallelism, *Advances in Neural Information Processing Systems*, Vol.32, pp.103–112 (2019).

[2] Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N.R., Ganger, G.R., Gibbons, P.B. and Zaharia, M.: PipeDream: Generalized pipeline parallelism for DNN training, *Proc. 27th ACM Symposium on Operating Systems Principles*, pp.1–15 (2019).

[3] Guan, L., Yin, W., Li, D. and Lu, X.: XPipe: Efficient pipeline model parallelism for multi-GPU DNN training, arXiv preprint arXiv:1911.04610 (2019).

[4] Kim, C., Lee, H., Jeong, M., Baek, W., Yoon, B., Kim, I., Lim, S. and Kim, S.: torchgpipe: On-the-fly pipeline parallelism for training giant models, arXiv preprint arXiv:2004.09910 (2020).

[5] Yang, B., Zhang, J., Li, J., Ré, C., Aberger, C. and De Sa, C.: Pipemare: Asynchronous pipeline parallel DNN training, *Proc. Machine Learning and Systems*, Vol.3 (2021).

[6] Chen, T., Xu, B., Zhang, C. and Guestrin, C.: Training deep nets with sublinear memory cost, arXiv preprint arXiv:1604.06174 (2016).

[7] Real, E., Aggarwal, A., Huang, Y. and Le, Q.V.: Regularized evolution for image classifier architecture search, *Proc. AAAI Conference on Artificial Intelligence*, Vol.33, No.1, pp.4780–4789 (2019).

[8] Ronneberger, O., Fischer, P. and Brox, T.: U-net: Convolutional networks for biomedical image segmentation, *International Conference on Medical Image Computing and Computer-assisted Intervention*, pp.234–241, Springer (2015).

[9] Tanaka, M., Taura, K., Hanawa, T. and Torisawa, K.: Automatic Graph Partitioning for Very Large-scale Deep Learning, *CoRR*, Vol.abs/2103.16063 (2021) (online), available from ⟨https://arxiv.org/abs/2103.16063⟩.

[10] Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K.: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, *Proc. 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1* (*Long and Short Papers*), pp.4171–4186, Association for Computational Linguistics (online), DOI: 10.18653/v1/N19-1423 (2019).

[11] He, K., Zhang, X., Ren, S. and Sun, J.: Deep Residual Learning for Image Recognition, *2016 IEEE Conference on Computer Vision and Pattern Recognition* (*CVPR*), pp.770–778 (online), DOI: 10.1109/CVPR.2016.90 (2016).

[12] Kolesnikov, A., Beyer, L., Zhai, X., Puigcerver, J., Yung, J., Gelly, S. and Houlsby, N.: Large Scale Learning of General Visual Representations for Transfer, *CoRR*, Vol.abs/1912.11370 (2019) (online), available from ⟨http://arxiv.org/abs/1912.11370⟩.

[13] Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J. and Catanzaro, B.: Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, *CoRR*, Vol.abs/1909.08053 (2019) (online), available from ⟨http://arxiv.org/abs/1909.08053⟩.

[14] Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y. and He, K.: Accurate, large minibatch SGD: Training ImageNet in 1 hour, arXiv preprint arXiv:1706.02677 (2017).

[15] Akiba, T., Suzuki, S. and Fukuda, K.: Extremely large minibatch SGD: Training ResNet-50 on ImageNet in 15 minutes, arXiv preprint arXiv:1711.04325 (2017).

[16] Kasagi, A., Tabuchi, A., Yamazaki, M., Honda, T., Miwa, M., Fukumoto, N., Tabaru, T., Ike, A. and Nakashima, K.: An Efficient Technique for Large Mini-batch Challenge of DNNs Training on Large Scale Cluster, *Proc. 29th International Symposium on High-Performance Parallel and Distributed Computing*, pp.203–207 (2020).

[17] Tokui, S., Okuta, R., Akiba, T., Niitani, Y., Ogawa, T., Saito, S., Suzuki, S., Uenishi, K., Vogel, B. and Yamazaki Vincent, H.: Chainer: A deep learning framework for accelerating the research cycle, *Proc. 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp.2002–2011 (2019).

[18] Akiba, T., Fukuda, K. and Suzuki, S.: ChainerMN: Scalable distributed deep learning framework, arXiv preprint arXiv:1710.11351 (2017).

[19] Krizhevsky, A., Hinton, G., et al.: Learning multiple layers of features from tiny images (2009).

滝澤 尚輝 （学生会員）

2020 年東京工科大学コンピュータサイエンス学部卒業．同大学大学院修士課程在学中．

矢崎 俊志 （正会員）

2007 年電気通信大学博士後期課程修了．2009 年東京工科大学助教．2010 年電気通信大学情報基盤センター助教．2012 年オハイオ州立大学 Visiting Scholar．2016 年一橋大学情報基盤センター助教．2018 年電気通信大学情報基盤センター特任准教授．2021 年電気通信大学同准教授．コンピュータアーキテクチャ，HPC，インターネット運用技術に関する研究に従事．博士（工学）．電子情報通信学会，ACM，IEEE 各会員．

石畑 宏明 （正会員）

1980 年早稲田大学理工学部卒業．同年（株）富士通研究所入社．画像処理システム，並列コンピュータの開発に従事．2007 年東京工科大学教授．並列コンピュータアーキテクチャの研究に従事．1992 年元岡賞，1993 年電子情報通信学会論文賞．博士（工学）．電子情報通信学会，人工知能学会，IEEE 各会員．