Vol.2021-HPC-180 No.9
2021/7/20

GPUクラスタにおけるハイブリッド並列DNN学習のボトルネック分析と改良

細木 隆豊[1]　野村 哲弘[1]　遠藤 敏夫[1]

概要：深層学習は計算量やメモリ使用量の観点から複数のアクセラレータを用いた分散化が注目されており，近年ではデータ並列とモデル並列を組み合わせたハイブリッド並列の研究が多くなされている．本論文ではハイブリッド並列をより効率的に活用するために，学習のボトルネックについて分析を行った．それにより，通信のオーバーヘッドやロードバランスが学習時間に影響をもたらすことがわかった．通信のオーバーヘッドの影響を小さくするように，GPU における計算と通信の並列性を上げ，通信を行うスレッドをまとめる改良を行った．その結果，データ並列性の利用に不均等性がある場合に最大 16%の学習時間の短縮が確認された．

1. はじめに

近年，Deep Neural Network(DNN) による深層学習は画像分類や自然言語処理，動画認識などの分野にて多大な進歩をもたらし，その他の分野への応用も拡大している．それに伴い計算量は増加しており，複数のアクセラレータによる並列化が必要とされている．DNN 学習の分散化方法としてデータ並列とモデル並列が一般的であるが，近年ではそれらを組み合わせたハイブリッド並列を活用する研究が盛んになされている．

本論文では，PipeDream でのハイブリッド並列深層学習におけるボトルネックの分析および改良を行った．PipeDream は，ミニバッチに対するパイプライン化と独自のモデル分配アルゴリズムを用いハイブリッド並列を効率的に活用する分散深層学習ライブラリである．PipeDream での学習について分析・実験を行うことで，通信に関するオーバーヘッドとモデルの分配が学習時間に影響することがわかった．

改良では，通信に関するオーバーヘッドの影響を小さくするように，GPU 上の通信と計算の並列性を上げ，通信を行うスレッドをまとめた．その結果，最大 16%の学習時間の短縮を可能にした．

2. 深層学習の分散化手法

本節では，深層学習の分散化・並列化のために利用される手法について説明する．

[1] 東京工業大学
Tokyo Institute of Technology

2.1 データ並列

データ並列は深層学習を高速化させるアプローチとして一般的に用いられている並列化手法で，データセットをいくつかのサブセットに分割し各プロセスに割り当てる．全てのプロセスはパラメータの複製を保持するため，各プロセスは割り当てられたデータに対して独立に順伝播・逆伝播を行う．その後，Allreduce 通信によりパラメータの勾配を平均化し，パラメータを更新する．

2.2 モデル並列

モデル並列はモデルを分割して各プロセスに割り当て，全てのプロセスで 1 つのモデルを学習する分散化手法である．各プロセスは前のプロセスからデータを受け取り，割り当てられたモデルの一部に対する計算を行ったのち，次のプロセスへ結果を送信する．各プロセスがモデルパラメータの一部のみを保持するためメモリ使用量が低くなり，アクセラレータのメモリ容量を超える大規模なモデルの学習を可能にする．

モデル並列において，モデルの分割によって得られる連続する層の集合をステージと呼ぶ．

2.3 ハイブリッド並列

ハイブリッド並列はデータ並列とモデル並列を組み合わせた新たな分散学習手法として，近年多くの研究がなされている．ハイブリッド並列の活用により，データ並列を超えるほどの高速化や大規模なモデルの学習の高速化が期待される．

本研究で対象とした PipeDream はハイブリッド並列を

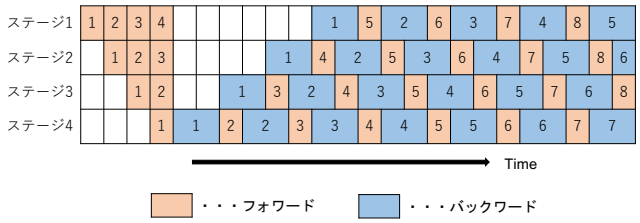

図 1　PipeDream におけるパイプライン化モデル並列の処理の概要図

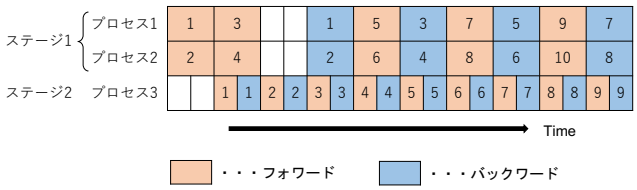

図 2　パイプライン化ハイブリッド並列の処理の概要図

活用する．

3. PipeDream における分散深層学習の実装

本節では，本研究で調査・改良の対象とした PipeDream[1] について説明する．PipeDream は，Deepak Narayanan らが提案した深層学習の GPU 分散化ライブラリであり，パイプライン化によってモデル並列・ハイブリッド並列を効率的に活用する．

3.1 パイプライン化モデル並列

パイプライン化はモデル並列を効率的に実現する並列化手法である．本手法では学習データを次々と投入しパイプライン処理をすることで，計算をオーバーラップさせハードウェア利用率を上げる．PipeDream におけるパイプライン化モデル並列の処理を**図 1** に示す．PipeDream のパイプライン化モデル並列では順伝播と逆伝播の両方で効率的にパイプライン処理を行うため，各プロセスはフォワードとバックワードの処理を交互に行う．

3.2 パイプライン化ハイブリッド並列

中間出力や勾配を伝播していくという深層学習の逐次的な性質により，パイプライン化モデル並列における学習時間は最も時間のかかるステージに依存する．そのため，ステージへの適切な分割が求められる．しかし，各層の処理にかかる時間の分散は大きく，モデルの分割を常に均等にできるとは限らない．

この問題を緩和するために，PipeDream はパイプライン化モデル並列とデータ並列を組み合わせたパイプライン化ハイブリッド並列を提案した．パイプライン化ハイブリッド並列の処理を**図 2** に示す．この手法では，各ステージを複数のプロセスに割り当てる．同じステージを担当するプロセスにはそれぞれ異なるミニバッチが割り当てられ，

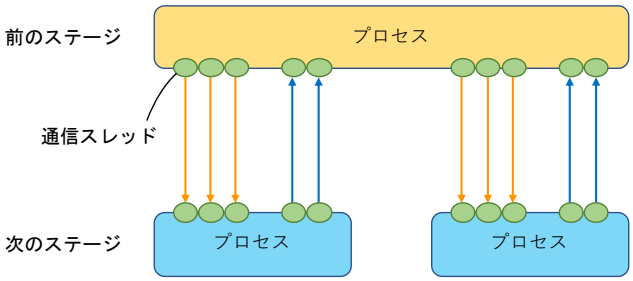

図 3　ステージ間通信の概要図

データ並列により処理が行われる．そのため，ステージにおける 1 ミニバッチの計算のレイテンシは変わらないがスループットは向上し，各ステージの処理時間の分散を小さくすることができる．

3.3 モデル分配アルゴリズム

パイプライン化モデル並列・ハイブリッド並列による学習を効率的に行うためには，デバイスへのモデルの分配が非常に重要となる．この決定にはモデルだけでなく，ハードウェアの構成にも強く依存する．

PipeDream では，最適なモデルの分配を行うために独自のモデル分配アルゴリズムを使う．このアルゴリズムはモデルの情報やマシンの構成（GPU メモリ容量，ノード内 GPU 数，ノード数，ノード間・ノード内の通信速度）を入力とし，動的計画法によって最適な決定を行う．

3.4 通信

PipeDream での学習では，ステージ内通信とステージ間通信の二種類の通信が行われる．

ステージ間通信は，中間出力と正解ラベルまたは勾配を転送するために用いられる．この通信は，次に処理を行うプロセスとの 1 対 1 通信でなされる．実装は PyTorch[2] の `distributed` ライブラリの `broadcast` 関数が使われている．また，PipeDream は通信用のスレッドを立ち上げることによって計算とオーバーラップさせている．**図 3** に通信スレッドの概要を示す．通信スレッドは，相手となるプロセスや送受信，データの種類毎に立ち上げる．

ステージ内通信は，複製間で勾配を平均化する際に用いられる．勾配の平均化は，ステージを担当する全プロセスで集団通信により実行される．分散データ並列学習をサポートする PyTorch の `DistributedDataParallel` ライブラリを用いて実装されており，バックワード中に勾配が平均化される．

また，PyTorch では GPU 間通信を行う通信バックエンドとして，Gloo[3] と NCCL[4] がサポートされている．NCCL は GPU 間の直接通信を行うライブラリである．GPU 間インターコネクトを通して高スループットと低レイテンシを実現できる．しかし，一つのデバイスに対して複数のスレッドから通信を呼び出すとハングアップする可能性があ

表 1　TSUBAME3.0 の構成

CPU	Intel Xeon E5-2680 v4　2.4GHz x2CPU
コア数/スレッド	14 コア / 28 スレッド x2CPU
CPU メモリ容量	256GiB
GPU	NVIDIA TESLA P100 x4
GPU メモリ容量	16GiB/GPU
CPU-GPU 間接続	PCI-Express Gen3 x16 (16GB/s)
SSD	2TB
インターコネクト	Intel Omni-Path HFI (100Gbps x4)

表 2　PipeDream の各段階における処理内容の一覧

receive_for	フォワードに必要なテンソルの通信待ち
run_for	フォワードの計算
send_for	中間出力を送信用スレッドへ送信
receive_back	バックワードに必要なテンソルの通信待ち
run_back	バックワードの計算
_sync	複製間の同期（run_back 中に実行）
send_back	勾配を送信用スレッドへ送信
_send	送信を行う関数の呼び出し
comp_send	送信を行う関数の終了
_recv	受信を行う関数の終了
send for forward	中間出力の通信
send for backward	勾配の通信

り，現在の実装ではハイブリッド並列の際に利用できない．そのため，本研究では Gloo を用いている．Gloo を用いた通信では，CPU 間で通信した後に GPU へデータをコピーする．NCCL と比較するとレイテンシが高く，スループットも低くなってしまう．

4. PipeDream のボトルネック分析

4.1 実験環境

本研究における実験は，東京工業大学学術国際情報センターの TSUBAME3.0 を利用した．TSUBAME3.0 における計算ノードの一台あたりの基本スペックを**表 1** に示す [5].

また，1 プロセスに 1GPU を対応させる．

4.2 実行時のボトルネック分析

PipeDream での学習におけるボトルネックを分析する．この分析では実行をいくつかの段階に分け，それぞれの段階に対する処理時間を計測した．**表 2** に各段階における処理内容の一覧をまとめる．テンソルの送信を行う関数では始めに受信側のメモリ確保のためテンソルの形状を送信し，その後テンソルを送信する．

Alexnet の分析から作成したタイムラインの一部を**図 4** に示す．この実行におけるモデルの分配では，3 プロセスのステージと 1 プロセスのステージがある．この図を見ると，プロセス 3 では計算を行なっていない箇所が存在する．この実装では計算と通信に単一の CUDA stream を設定しており，GPU 上で計算と通信の並列性が十分に活用されていないことが原因と予想される．

また，プロセス 3 はプロセス 0,1,2 からの通信が集中することも確認された．この実装におけるプロセス 3 の通信スレッドの数は 9 つであり，これらのうち複数のスレッドが同時に動作することがわかった．

4.3 モデル分配アルゴリズムに対する評価

本研究ではモデル分配アルゴリズムにパラメータとして与えるマシンの構成に対していくつかの条件を用意し，それぞれの実行時間を比較した．具体的にはインターコネクトのバンド幅として，理論バンド幅を与える条件や実測値に基づいた条件，計算時間のみで最適な決定を行うようにバンド幅を極端に大きく設定した条件を用意した．

この比較から，どのモデル・ノード数においても常に最適なモデル分配を行う条件は存在しないことがわかった．この結果は，PipeDream のモデル分配アルゴリズムが学習のボトルネックとなりうることを示唆しており，また最適化の余地があることがわかる．

5. 改良の提案

5.1 提案

4.2 節での分析をもとに，通信のオーバーヘッドを小さくするように二つの改良を提案する．

提案 1　GPU における計算と通信の並列性を上げる．

提案 2　通信スレッドをまとめる．

提案 1 は通信待ちのために計算が行われていない箇所が存在することに対する改良である．元の実装において計算と通信に同じ CUDA stream が設定されていることが確認できているため，この提案は有効であると考えられる．提案 2 については複数の通信スレッドが同時に動作していることに対する改良となる．複数の通信を別スレッドから同時に呼び出す場合と単一スレッドから呼び出した場合で，API 通信の最適化に違いがある可能性があり，それによる通信への影響を考慮している．

5.2 実装

提案 1 および提案 2 の両方の改善を行った実装を行った．提案 2 について 2 通りの実装方針が考えられたため，それぞれ impl-SA および impl-SRA として実装した．

提案 1 については，エポック開始時に各プロセスで `torch.cuda.Stream()` によって新たに CUDA stream を生成し，そのプロセスにおける全てのステージ間通信でこの stream を指定する．

提案 2 に関してはそれぞれの実装で異なる．以下で詳細について述べる．

また，これらの実装は学習率に影響する箇所についての改良を行っておらず，元の実装と比較した際の学習率への影響はない．

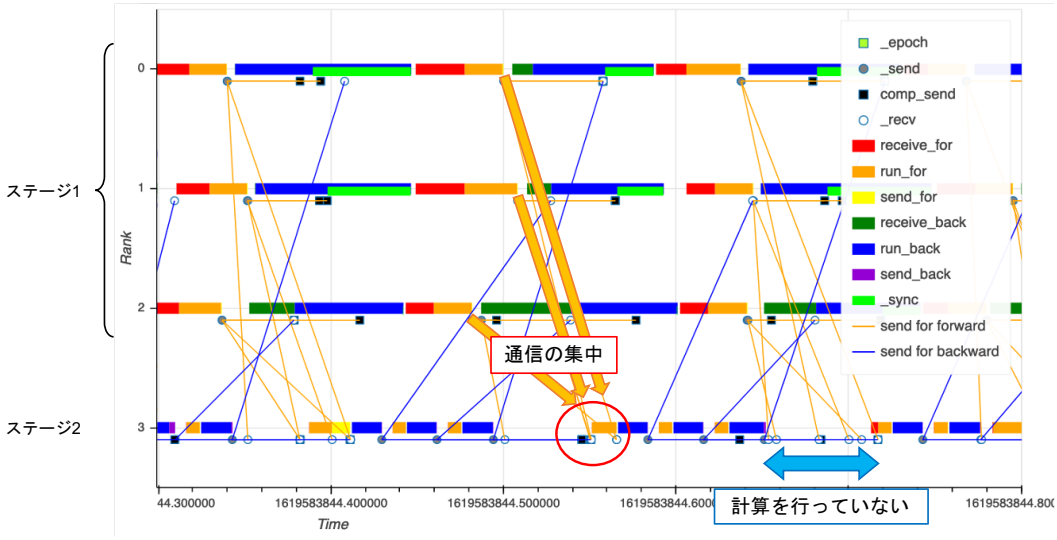

図 4　元の実装での実行時タイムライン（Alexnet）

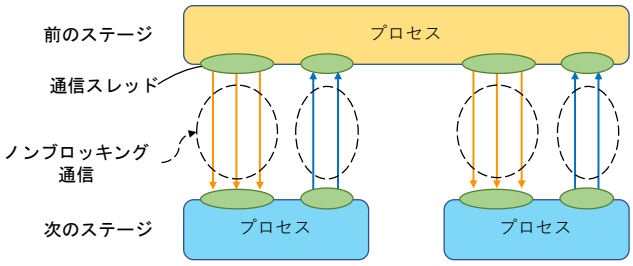

図 5　impl-SA の概要図

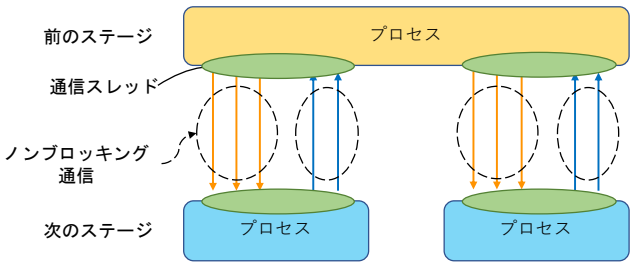

図 6　impl-SRA の概要図

5.2.1　impl-SA

この実装での通信は図 5 のようになる．この実装では，相手となるプロセスと送受信別に通信スレッドを立ち上げる．また，点線で囲まれた通信は同じミニバッチの異なるデータに対する通信で，これら全てのデータが次のステージでの処理の入力となる．そのため，同時に通信されることが望ましい．

本実装ではノンブロッキング通信を用いることで同時に通信されるようにした．`torch.distributed.broadcast` 関数には非同期操作を行うためのオプション引数があり，それによってノンブロッキング通信を実現した．この非同期操作は終了を待つ同期操作を実行する必要がある．本実装では，1 つのミニバッチに関する全てのテンソルで通信を呼び出したのちに，呼び出した順に同期する．

表 3　実験に用いた画像数とバッチサイズ

	Alexnet	ResNet50	VGG16
画像数	122880	24576	15360
バッチサイズ	256	32	16

5.2.2　impl-SRA

この実装での通信は，図 6 のように相手となるプロセス毎に通信スレッドを立ち上げる．送信と受信を同じスレッドで実行するため，これらの実行順序に注意しなければならない．そのため，エポック開始時にステージ間通信のスケジュールを作成し，通信スレッド間での通信順序の一貫性を保っている．

また，この実装においても impl-SA と同じく非同期操作を用いて，点線で囲まれた通信はノンブロッキング通信での実装を行った．

6.　実験と評価

6.1　実験の前提

実験では，Alexnet[6]，ResNet50[7]，VGG16[8] を学習モデルとして用いた．それぞれのモデルの学習で用いたデータセットの画像数とバッチサイズを表 3 に示す．ただし，データセットには画像サイズ $3 \times 224 \times 224$ となるようにランダムに生成した合成データを用いている．

本節の実験は 4.1 節と同じ実験環境で行い，またノード数は 1,2,4,8（GPU 数は 4,8,16,32）として行っている．

それぞれのモデルやノード数におけるモデルの分配は，元の実装において最も実行時間が短かった決定で比較を行っている．

6.2　改良の評価

提案した改良と元の実装での実行時間を図 7 に，元の実装での実行時間に対する相対実行時間を表 4 に示す．

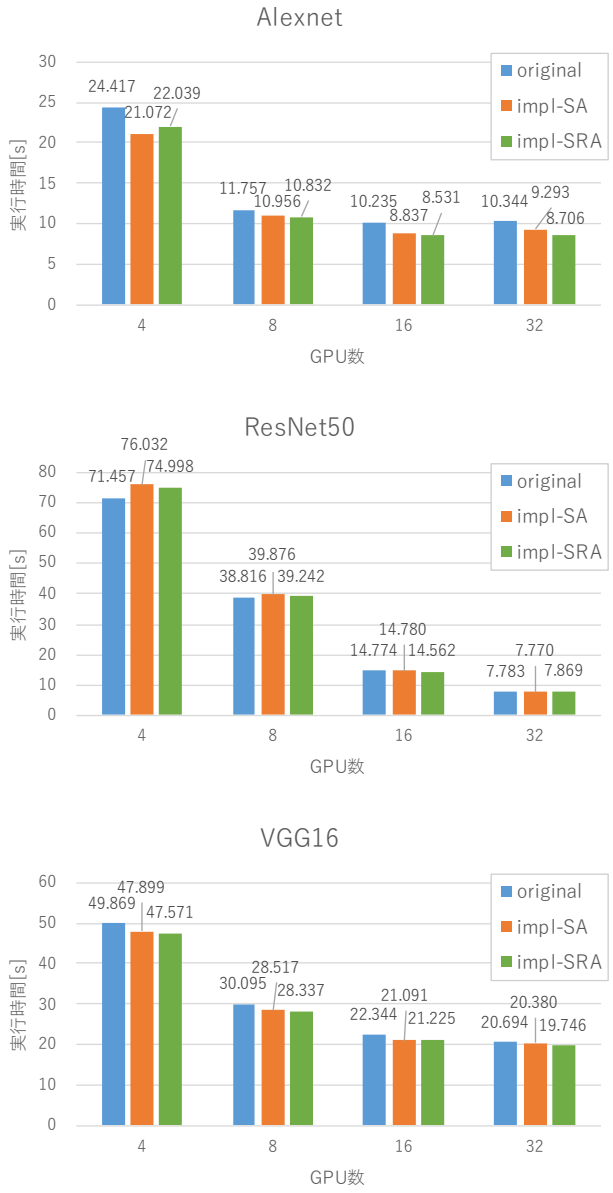

図 7　改良の比較

表 4　original に対する改良の相対実行時間 [%]

Alexnet

GPU 数	impl-SA	impl-SRA
4	86.3	90.3
8	93.2	92.1
16	86.3	83.4
32	89.8	84.2

ResNet50

GPU 数	impl-SA	impl-SRA
4	106.4	105.0
8	102.7	101.1
16	100.0	98.6
32	99.8	101.1

VGG16

GPU 数	impl-SA	impl-SRA
4	96.0	95.4
8	94.8	94.2
16	94.4	95.0
32	98.5	95.4

表 5　Alexnet の学習時の通信時間 [ms]

		original	impl-SA	impl-SRA
out0 (42.25 MiB)	0→3	68.7	59.7	47.9
	1→3	67.3	59.5	47.4
	2→3	73.0	60.9	50.2
	3→0	62.1	68.6	49.8
	3→1	68.5	72.3	49.1
	3→2	59.9	69.5	51.5

表 6　ResNet50 の学習時の通信時間 [ms]

		original	impl-SA	impl-SRA
out0 (49 MiB)	0→2	48.1	51.0	58.8
	1→3	48.7	50.4	59.1
	2→0	44.6	46.9	44.8
	3→1	44.8	46.3	44.8
out1 (12.25 MiB)	0→2	25.6	50.1	45.9
	1→3	25.5	49.4	42.4
	2→0	24.4	44.7	42.5
	3→1	24.8	44.0	42.4

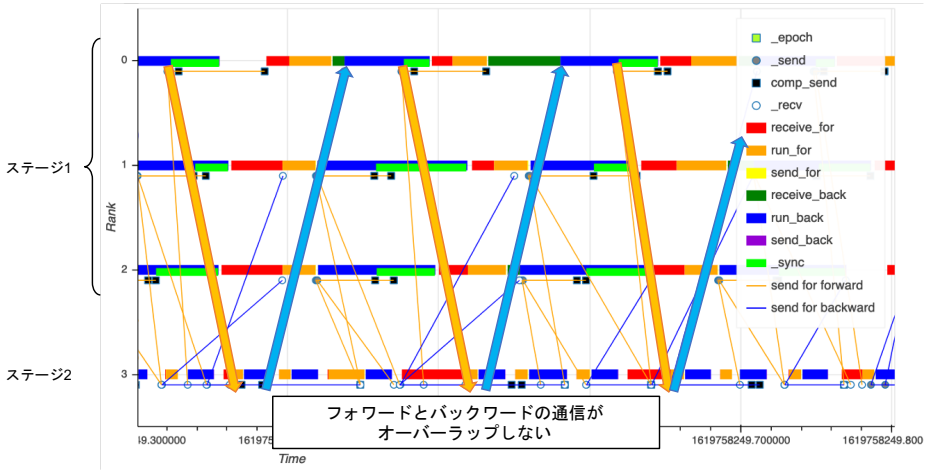

図 8　impl-SRA での実行時タイムライン（Alexnet）

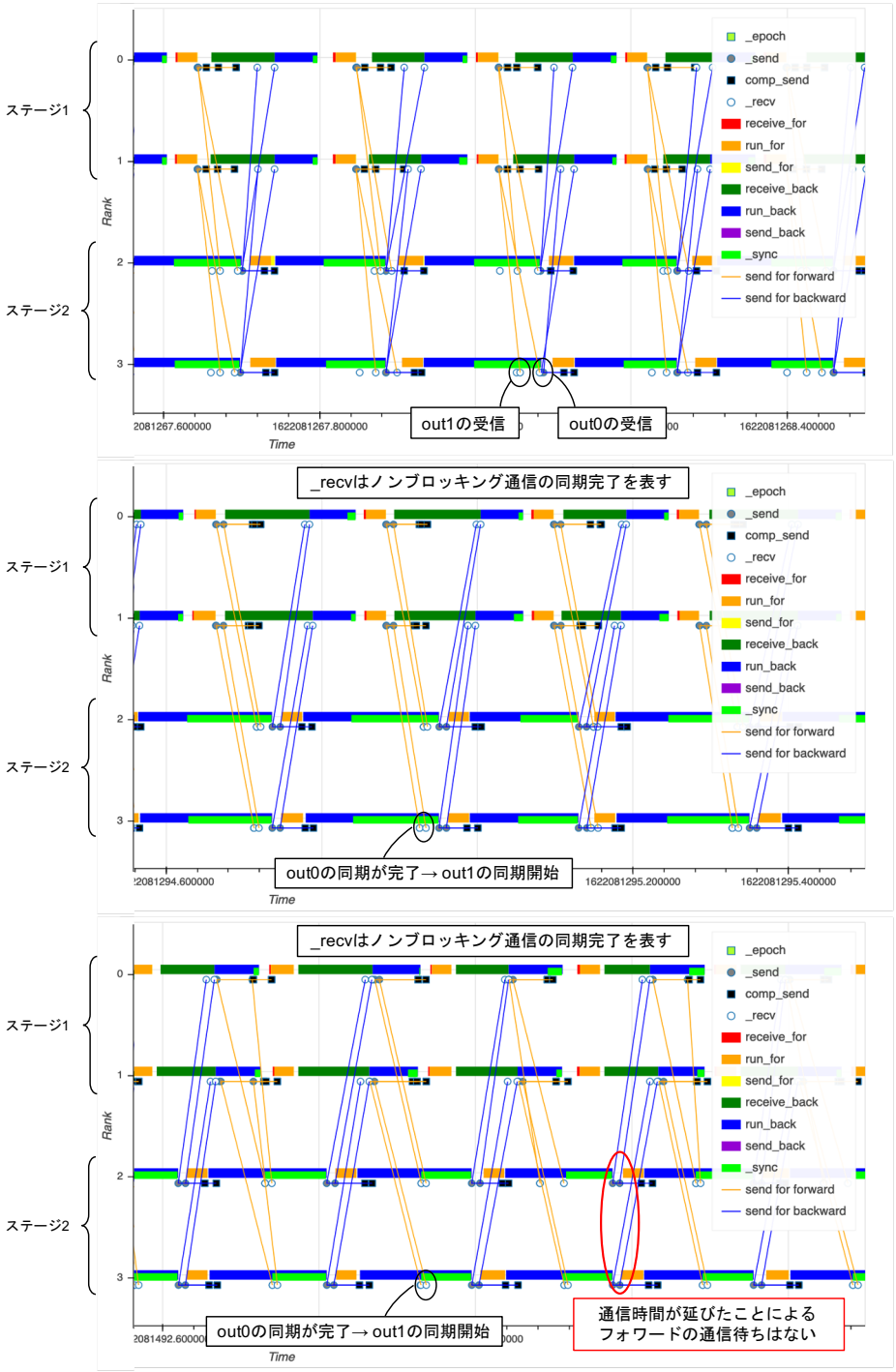

図 9 original（上），impl-SA（中），impl-SRA（下）での実行時タイムライン（ResNet50）

この結果を見ると，Alexnet と VGG16 の学習に対しては impl-SA と impl-SRA の両方で実行時間が短縮した．特に Alexnet では最大 16%の実行時間の短縮が確認された．

一方で，ResNet50 では改善が見られず，学習時間の増加も確認される．GPU 数が 16，32 の場合では，データ並列性のみを活用しステージ間通信が行われないため，今回の改良による影響がない．GPU 数が 4，8 の場合では，ステージ数が 2 つでそれぞれのステージを同じ数のデバイスへ割り当てており，これが原因の一つと考えられる．ミニバッチの割り当てが Round Robin 方式に従うことで，各プロセスのステージ間通信が特定のプロセスとのみで全て行われる．そのため，複数のプロセスから一つのプロセスへ通信が集中する状況が発生せず，提案 2 による影響が小さかったと考えている．

6.3 通信に対する評価

改良が通信に与えた影響について考察する．4GPU での実行時の`_send` から`_recv` までにかかる平均時間を比較する．

Alexnet の学習における平均通信時間を**表 5** に示す．

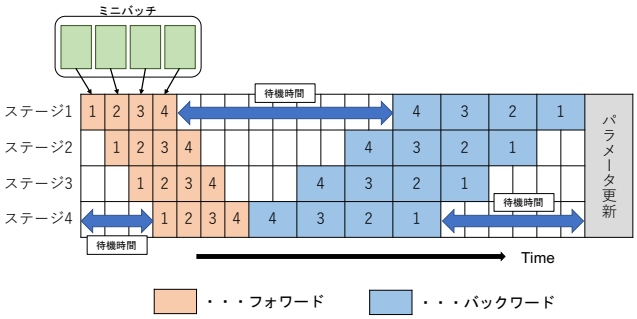

図 10　GPipe におけるパイプライン化モデル並列の処理の概要図

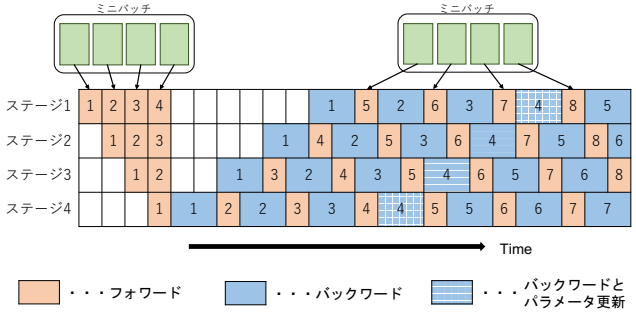

図 11　PipeDream-2BW におけるパイプライン化モデル並列の処理の概要図

impl-SA ではフォワードでの通信（0,1,2→3）の時間短縮が確認されるが，一方でバックワードでの通信（3→0,1,2）では劣化が確認される．この劣化の原因については現在特定できていない．

impl-SRA ではどの通信においても通信時間の短縮が確認される．しかし，図 7 を見ると学習時間の短縮では impl-SA に劣っている．**図 8** の実行時のタイムラインを見ると，送信と受信のスレッドをまとめたことにより，フォワードとバックワードの通信がオーバーラップしないようになっている．そのため，通信の待ち時間が増えたことが起因していると考えられる．

次に，ResNet50 について考察する．ResNet50 での平均通信時間を**表 6** に示す．また実行時タイムラインを**図 9** に示す．out1 についてはどちらの実装においても大きく通信時間が延びている．この原因として，改良での`_recv`がノンブロッキング通信の同期が完了した時点を表しており，out1 の同期の呼び出しが out0 のノンブロッキング通信の同期が完了したのちに起こることが考えられる．これによって，out1 の通信時間が out0 の通信時間に影響されている．

out0 については，impl-SA では通信時間は大きな変化は確認されない．一方，impl-SRA ではフォワードの通信（0→2 と 1→3）が大きく悪化していることがわかる．図 9（下）を見ると，フォワードの通信時間が延びたことによる通信待ちは確認できない．そのため，通信時間の増加が学習時間の増加の原因とはならないと予想される．

改良による通信時間の増加の原因と共に，学習時間の増加の原因について今後も調査を行っていこうと考えている．

7. 関連研究

7.1 GPipe

パイプライン化モデル並列を利用し大規模モデルの学習を効率的に行う分散化手法として，Huang らは GPipe を提案している [9]．GPipe におけるパイプライン化モデル並列の処理の概要を**図 10** に示す．GPipe はミニバッチをさらに分割したマイクロバッチをパイプライン化をすることで，バッチサイズを変えずに時間を短縮する．GPipe は PipeDream の実行スケジュールと異なり，ミニバッチ内の全てのマイクロバッチにて順伝播を終えたのちに逆伝播を行い，その後パラメータを更新する．そのため，PipeDream に比べハードウェア利用率が低くなる．また，GPipe は現在データ並列を活用しない実装となっている．

GPipe ではステージ内の中間出力をバックワード中に再び計算する再計算という手法によって，中間状態を保持する必要がなくなり各プロセスのメモリ使用量を抑える．この再計算手法は，PipeDream においても有効であると考えられ，メモリ使用量を減少させるための拡張として期待される．

7.2 PipeDream-2BW

PipeDream を改善しメモリ使用量を減らしたパイプライン化ハイブリッド並列学習を行うシステムとして，Narayanan らは PipeDream-2BW を提案している [10]．PipeDream-2BW におけるパイプライン化モデル並列の処理の概要を**図 11** に示す．PipeDream-2BW は，PipeDream と異なりマイクロバッチに対するパイプライン化を行う．また，各ステージが保持するパラメータバージョンを 2 つに制限する double-buffered weight update (2BW) 機構を使い，高いハードウェア利用率と低いメモリフットプリントの両立を可能にする．

PipeDream-2BW では，モデルの分割数と複製数を指定することでハイブリッド並列を活用する．これにより各ステージのデータ並列性の利用が均等となり，本研究の提案 2 による学習時間の短縮は起こらないと推測される．一方，提案 1 は適用可能であり，通信のオーバーヘッドの削減に有効であると予想される．

8. まとめ

GPU クラスタ上でのハイブリッド並列による深層学習の高速化に向け，一例として PipeDream での学習に対しボトルネックの分析を行った．

その結果，通信に関するオーバーヘッドとモデル分配アルゴリズムがボトルネックとなりうることがわかった．このうち，通信に関するオーバーヘッドを緩和するために，

GPU における計算と通信の並列性を上げ，通信を行うスレッドをまとめるように改良を加えた．これらの提案をもとに 2 通りの実装を行った．その結果，Alexnet および VGG16 に対する実験では，どちらの実装においても学習時間が短縮し，その減少率は最大 16%であった．

今後は，改良によって通信時間の増加や学習時間の増加が起こった原因について調査していこうと考えている．また，PipeDream のモデル分配アルゴリズムについて引き続き調査を行い，より適切なモデルの分配を決定するようモデル分配アルゴリズムの改良を考えている．他にも，現在の実装で使用されている通信バックエンドでは GPU 間直接通信がサポートされていないため，通信バックエンドの変更ができるか調査していく．GPU 間直接通信を実現することで，更なる高速化が期待される．

謝辞 本研究は、東京工業大学のスーパーコンピュータ TSUBAME3.0 を利用して実施した。また，本研究の一部は JSPS 科研費 20H04165 の助成による．

参考文献

[1] Narayanan, D. et al.: PipeDream: generalized pipeline parallelism for DNN training, *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 1–15 (2019).
[2] PyTorch: PyTorch, https://github.com/pytorch/pytorch.
[3] Facebook: Gloo, https://github.com/facebookincubator/gloo.
[4] NVIDIA: NCCL, https://github.com/NVIDIA/nccl.
[5] 松岡 聡ほか: HPC とビッグデータ・AI を融合するグリーン・クラウドスパコン TSUBAME3. 0 の概要，研究報告ハイパフォーマンスコンピューティング (HPC)，Vol. 2017, No. 29, pp. 1–6 (2017).
[6] Krizhevsky, A.: One weird trick for parallelizing convolutional neural networks, *arXiv preprint arXiv:1404.5997* (2014).
[7] He, K. et al.: Deep residual learning for image recognition, *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778 (2016).
[8] Simonyan, K. et al.: Very deep convolutional networks for large-scale image recognition, *arXiv preprint arXiv:1409.1556* (2014).
[9] Huang, Y. et al.: Gpipe: Efficient training of giant neural networks using pipeline parallelism, *Advances in neural information processing systems*, pp. 103–112 (2019).
[10] Narayanan, D. et al.: Memory-efficient pipeline-parallel dnn training, *arXiv preprint arXiv:2006.09503* (2020).