Vol.2019-HPC-168 No.26
2019/3/7

大規模メニーコアシステムOakforest-PACSにおけるディープラーニングの性能評価

田村 紘平[1]　塙 敏博[2,1]

概要：近年、深層学習のトレーニングにおいて，構成するネットワークが複雑になり，学習に用いるデータセットも増大していく中で，多数の GPU を搭載した大規模 GPU クラスタによる高速な学習手法が提案されている．一方で，CPU 中に多数のコアを搭載したメニーコアプロセッサも，ディープラーニングの基本演算に適した SIMD 演算ユニットを備えており，これらを導入したクラスタを用いて大規模な学習環境が実現できる．本研究では，深層学習フレームワークである ChainerMN を、メニーコアプロセッサ Intel Xeon Phi を搭載した Oakforest-PACS に適用し，性能評価を行い，通信性能に関する高速化を実施した．その結果，従来の通信手法と比べて約 2.1 倍の高速化を達成した．

1. はじめに

Deep Learning(以下 DL) は Deep Neural Network と呼ばれる計算モデルを用いた機械学習であり，近年注目されている人工知能（AI）を支える技術である．しかし，DL において高い精度を得るためには膨大な教師データを必要とする上に，複雑化したモデルを使用する必要があるため，それに伴ってパラメータの数も増大し続けている．そのため．GPU を効率よく利用してトレーニングすることが標準的なアプローチとして採用されてきたが，高い演算性能を誇る最新の GPU を使用した場合でも非常に長い時間を要する．そこで高速化の手法として Message Passing Interface（MPI）等を利用して複数の GPU を用いて並列分散処理をする方法が挙げられる．しかし，GPU のようなアクセラレータは，CPU と比べてオンチップメモリの容量に限りがあるといった問題が挙げられる．したがって，今後ますます複雑化していくモデルや大規模なデータセットを扱う場合，GPU を用いた場合でも，CPU メモリへのアクセスやノードを超えた GPU 間の通信が頻繁に発生し，トレーニング時間に影響を及ぼす可能性がある．

そこでメニーコアプロセッサを用いて，DL のトレーニングをする方法が挙げられる．メニーコアプロセッサとは，十数個から数百程度のコアを集積することで処理速度の向上を図ったマイクロプロセッサーである．そのようなメニーコアプロセッサにおいても DL の基本演算に向いた演算ユニットを備えており，高速化が期待できる．

本研究では，ユーザに広く使われている Chainer[1] をベースにし，比較的容易に並列分散学習を実現する ChinerMN[2] のメニーコアプロセッサへの適用を検討する．それによって，Intel Xeon Phi プロセッサを搭載した Oakforest-PACS（OFP）（最先端共同 HPC 基盤施設）を DL の大規模計算環境として提供することが可能になる．本稿においては，まず予備評価として OpenMP+MPI によるハイブリッド並列における実行時間を分析することで，プロセス数の増加に伴って通信時間が支配的になり，トレーニング時間に大きく影響することを確認した．そこで通信時間を短縮するために 2D-Torus All-Reduce[9] を基に，One-Sided 通信による通信の高速化を図った．

2. Deep Learning

2.1 学習プロセス

DL は以下の 3 つのステップを繰り返しながら最適な学習パラメータを決定するためにトレーニングを行う．

- Forward
  まず，入力層に入力ベクトル $\boldsymbol{x}$ が与えられる．入力層のユニット i にはベクトル $\boldsymbol{x}$ の第 i 次元目の値である x_i が入力される．そして x_i に重み $w_{ki}^{(1)}$ 乗じられて，中間層のユニット k に入力される．中間層のユニット k ではこれらの値を取り，さらにその和をある活性化関数 σ に与えた結果の値を出力する．したがって，一般に，第 l 層のユニット j の出力 $a_j^{(l)}$ は以下のような

[1] 東京大学大学院工学系研究科電気系工学専攻
Department of Electrical Engineering and Information Systems, Graduate School of Engineering, The University of Tokyo
[2] 東京大学情報基盤センター
Information Technology Center, The University of Tokyo

形で表される．

$$a_j^{(l)} = \sum_i w_{ji}^{(l-1)} \sigma(a_i^{l-1}) + b_j^{(l-1)}$$

入力層から出力層にかけて，全てのユニットに対してこの処理を行う．これを Forward 処理と呼ぶ．

- Backward
  Forward 処理で得られた出力ベクトル $\boldsymbol{f}$ と正解ベクトル $\boldsymbol{y}$ の 2 乗誤差である損失関数 E を以下のような形で求める．

$$E = \frac{1}{2}\sum_{j=1}^{n}(f_j - y_j)^2$$

この損失関数を最小化する学習パラメータの値を求めるために，各パラメータで偏微分を行う．この時に得られる $\nabla \boldsymbol{E}$ を勾配と呼ぶ．i 番目における学習パラメータベクトルを $\theta^{(i)}$，パラメータの数を V とすると勾配 $\nabla \boldsymbol{E}$ は以下の式で表される．

$$\nabla \boldsymbol{E} = \left(\frac{\partial E}{\partial \theta_1}\bigg|_{\theta=\theta^{(i)}}, \frac{\partial E}{\partial \theta_2}\bigg|_{\theta=\theta^{(i)}}, \cdots, \frac{\partial E}{\partial \theta_V}\bigg|_{\theta=\theta^{(i)}} \right)$$

この勾配を求める処理を Backward 処理と呼ぶ．

- Optimize
  Backward 処理で得られた勾配値を基に損失関数の最小値を求め，その時における学習パラメータの値を次のパラメータとして更新をする．Optimize のアルゴリズムにはいくつか種類があるが，最も基本的なアルゴリズムとして確率的勾配法（SGD: Stochastic Gradient Descent）が挙げられ，以下のように表される．α は学習率と呼ばれ，値が大きいほど更新量が大きくなる．

$$\theta^{(i+1)} = \theta^{(i)} - \alpha \nabla \boldsymbol{E}$$

2.2 モデル並列とデータ並列

大規模 DL は大きく分けて，モデル並列学習（図 1）とデータ並列学習（図 2）の 2 種類がある．それぞれの特徴についてまとめる．

モデル並列学習は学習処理を行う DNN を分割し，それぞれ別の演算器で処理を行う方法であり，演算器間を各層の入出力データが転送される．複数の演算器に一つの DNN を分割して処理するため，巨大な DNN を演算する手法として用いられるが，各演算器の処理量を一定にすることが難しくなる点がデメリットとして挙げられる．さらに，各処理において演算器間で転送されるデータの要素数にばらつきが生じやすく，バッチサイズにも比例して分散が増大する．そのためノード数の増加に伴い，特定のノードの演算や通信がボトルネックとなり性能が低下しやすい．

データ並列学習は各演算器が同じ DNN モデルを保持した状態で，それぞれが異なるバッチデータを用いて学習する方法である．学習処理は，各演算器で異なるバッチデータから算出された勾配情報 ∇E を集約し．パラメータの更新を行った上で次のバッチ処理を行う．モデル並列とは異なり，各演算器が独立して Forward 処理から Backward 処理を実行できるため，演算器は各層の処理を中断することなく実行できる．しかしデータ並列学習は，DNN の重みパラメータのサイズによって決まる ∇E を通信し集約するためパラメータサイズの大きい DNN の場合，集約処理の負荷が大きくなる特徴がある．

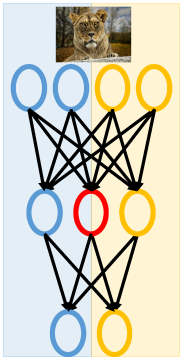

図 1　モデル並列学習

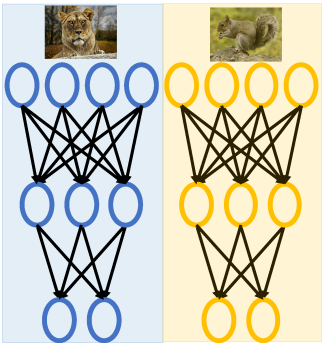

図 2　データ並列学習

また，データ並列学習は更に非同期型と同期型に分けられる．非同期型は各ワーカがそれぞれの速度で学習を進め，ワーカー間で同期はせずにそれぞれがパラメータサーバから勾配を得る方式である．一方で同期型データ並列学習は各ワーカが同時に処理を進めてく方式で Backward の後に得られた勾配の値を同時に通信をして集約をする方式である．

非同期型はスループットが高い一方で，Staleness と呼ばれる現象が起こる問題がある．Staleness は勾配を更新する際にすでに他のワーカーが勾配を更新したあとで，古いパラメータに対して更新された勾配を得ることになってしまうことであり，モデルの改善に悪影響を及ぼしてしまう．同期型は Staleness は起こらないが，通信による勾配の集約処理によってオーバーヘッドが発生してしまう問題がある．しかし，Staleness が起きないことや，チューニングのしやすさから現在は同期型データ並列学習による分散学習が主流になっている．

2.3 バッチサイズと精度の関係

DL におけるトレーニングでは実行時間の短縮だけでなく，トレーニングによって得られたモデルが精度の高い汎化性能を持つことが求められる．ここでは分散学習におけるモデルの精度について説明する．

データ並列学習においては各プロセスのミニバッチサイズにプロセス数を掛けた値が全体のバッチサイズに相当する．バッチサイズが大きくなると，エポック（学習において訓練データをすべて使い切るまでの期間）あたりのイテレーション回数が減ってしまうため，精度が劇的に劣

表 1　各研究報告における Imagenet/ResNet-50 の結果

	Batch Size	Processor	DL Library	Time	Accuracy
He et al.[4]	256	Tesla P100 x8	Caffe	29 hours	75.3%
Goyal et al.[5]	8K	Tesla P100 x256	Caffe2	1 hour	76.3%
Smith et al.[6]	8K → 16K	full TPU Pod	TensorFlow	30 mins	76.1%
Akiba et al. [7]	32K	Tesla P100 x1024	Chainer	15 mins	74.9%
Jia et al. [8]	64K	Tesla P40 x2048	TensorFlow	6.6 mins	75.8%
Mikami et al. [9]	34K → 68K	Tesla V100 x2176	NNL	224 secs	75.03%
Ying et al. [10]	32K	TPU v3 x1024	TensorFlow	132 secs	76.3%

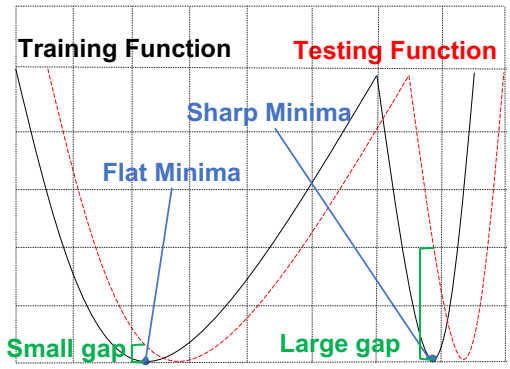

図 3　Flat Minima と Sharp Minima[3]

化してしまう．加えて，勾配の分散が小さくなることにより，質の悪い局所解に進みやすくなる．そのためプロセス数を増やしすぎると結果として得られるモデルの汎化性能が悪くなってしまうという現象も知られている．この時の局所解は Sharp Minima と呼ばれており，損失関数の局所解の中でも比較的尖った点を指す．一方で，局所解の中でも比較的平坦な点を Flat Minima と呼ぶ．データ並列のように巨大なバッチサイズで学習を行う場合，この Sharp Minima に収束しやすいことが指摘されている [3]．これは先述した通りバッチサイズが大きいと勾配の分散が小さくなり，損失関数において解の探索が局所化するためである．Flat Minima は、図 3 に示す様に局所解から多少パラメータを変動させても、ロスがあまり増加しないような解であり、訓練データとテストデータの分布の違いにより損失関数がずれた場合でも大きく精度が変わらないため，高い汎化性能が得られる．一方で Sharp Minima は局所解からパラメータが変動するとロスが大きく増加してしまうため，汎化性能が低くなってしまう．そのため，プロセス数をあげることによりモデルの汎化性能が悪くなってしまうと言える．

2.4　並列分散学習の関連研究

近年では複数の GPU を活用した分散学習によって大規模 DL のトレーニングを対象にした学習時間の短縮を行う研究が盛んに行われている．表 1 は，各研究報告における Top-1 正解率（推論において，最もスコア高かった答えが正解である割合）と学習時間を示しており，より短い時間で高い精度を得ることが可能になっていることがわかる．ベンチマークは，DL の分散学習速度を測る際に一般的に活用されている ImageNet/ResNet-50 を用いている．ImageNet とは 1000 万枚以上の画像を集め，分類したデータセットである．また，ResNet-50 とはある層で求める最適な出力を学習するのではなく，層の入力を参照した残差関数を学習することで高い精度が得られるモデルである．

Goyal らは巨大なバッチサイズで学習する際に，2.3 節で述べたように Sharp Minima に陥ることを防ぐために Linear Scaling Rule を提案している [5]．これはミニバッチサイズが k 倍になるときに学習率も k 倍するといったもので，パラメータの変動を大きくすることで Sharp Minima に収束することを防ぐ手法である．これにより 8K のバッチサイズにおいても 76.3% と高い精度のモデルを得ることが可能になった．また，秋葉らは RMSprop Warmup と呼ばれる手法によって，更に巨大な 32K のバッチサイズで学習を達成している．RMSprop Warmup は DL の最適化手法である Momentum SGD と RMSprop を組み合わせ，学習の初期段階は RMSprop を使用し，そこから徐々に Momentum SGD に切り替えていく手法である．これにより 15 分で 74.9% の精度を達成している．

また，データの集約をする際の通信を工夫することにより学習の高速化を図る研究例もある．三上らは通信を行うプロセス群を複数のグループに分けることでメッセージ長を分割し，All-Reduce における通信コストを削減することで通信の高速化を達成している [9]．その結果，224 秒で 75.03% の精度を達成している．この手法は 2D-Torus All-Reduce と呼ばれ，詳しくは 5.2 節で後述する．

3.　Oakforest-PACS

3.1　概要

本研究は計算機環境として Oakforest-PACS（以下 OFP）（最先端共同 HPC 基盤施設）を使用した．ここでは本システムについて説明する．

OFP の主な仕様を表 2 に示す．OFP は 8,208 台の計算ノードで構成されており，総理論演算性能（倍精度浮動小数点演算性能）は 25PFLOPS である．計算ノードは Intel Xeon Phi プロセッサ（開発コード名:KNL=Knights

表 2　Oakforest-PACS の主な仕様

Item		Spec.
Theoretical Computation Performance		25.004 PFLOPS
Number of Node		8,208
Total of Memory Capacity		897 TByte
Parallel File System	Number of Server	10
	Storage Capacity	26 PB
	Data Transfer Rate	500 GB/sec
High-speed Cache System	Number of Server	25
	Capacity	940 TB
	Data Transfer Rate	1,560 GB/sec

表 3　KNL の主な仕様

Item		Spec.
Operation Frequency		1.40 GHz
Theoretical Computation Performance		3046.4 GFLOPS
Interconnect		Intel Omnipath Architecture (100Gbps) Full-bisecxtion BW Fat-tree
Number of Core	Phyisical	68
	Logical	272
Storage Capacity	MCDRAM	16 GB
	DDR4	96GB
Memory Bandwidth	MCDRAM	490 GB/s(Effective Rate)
	DDR4	84.5 GB/s(Effective Rate)

Landing）1 ソケットで構成されており，それ自身がブート可能なメニーコアシステムである．KNL に関する詳しい説明は 3.2 節で後述する．

OFP の各計算ノードは Intel Omni-Path(12.5 GB/s) によりフルバイセクションバンド幅で接続され、ノード間通信とファイル共有を行う．並列ファイルシステムはアクセス性能向上のための高速ファイルキャッシュシステム持つ．並列ファイルシステムの有効利用容量は 26 PB，データ転送速度は 400 GB/s であり，高速ファイルキャッシュシステムの有効利用容量は 940 TB，データ転送速度は 1,560 GB/s である．

3.2　Knights Landing

続いて OFP に搭載されたメニーコアプロセッサ KNL について説明する．KNL の構成図を図 4，計算環境の概要を表 3 に示す．KNL は 1 タイルに 2 コア配置されており，2 コアが L2 データキャッシュを共有している形になっている．また，512 ビット幅のベクトルレジスター AVX-512 ユニットを各コアに 2 基備えており，DL のような大量のデータに対して同種の演算を行うような処理において高い性能を得ることができる．また、4-way ハイパースレッディングにより 68 物理コア $\times 4 = 272$ の論理コアを使用することが可能である．

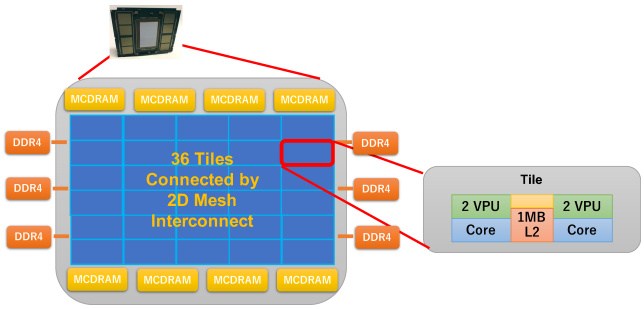

図 4　KNL の構成図

3.3　メモリモードとクラスタリングモード

メモリは，容量 16GB の広帯域メモリ MCDRAM を搭載しており，オフダイのメモリとして DDR4 を 6 チャネル接続している．MCDRAM は NUMA（Non-Uniform Memory Access）の動作の制御が可能であるため，複雑なメモリアクセスパターンを含むアプリケーションにおいても効率よく計算を行うことができる．この MCDRAM を利用するためのモードとして以下のメモリモード，クラスタリングモードを備えている．

メモリモード

- Flat: MCDRAM と DDR4 が独立したアドレス
- Cache: MCDRAM は DDR4 メモリのキャッシュとして動作
- Hybrid: MCDRAM の容量を分割して，Flat モードと Cache モードの両方を利用

クラスタリングモード

- All-to-all:アドレス情報が全体に分散
- Quadrant, Hemisphere:内部でアドレス情報が 4(または 2) に分割
- SNC-4, SNC-2:NUMA ドメインが明示的に 4(または 2) に分割

OFP においては，メモリモードについては，Flat と Cache それぞれに専用のキューを設けて使い分けられており，クラスタリングモードについては Quadrant モードで動作する．性能に関しては KNL の前世代の Intel Knights Corner から比較しても大幅に向上しており，演算性能は単精度で 2TFLOPS から 6TFLOPS，メモリバンド幅は STREAM ベンチマークにおいて 159 GB/s から 450GB/s と大幅に改善されている．

4.　予備評価と分析

本研究では Oakforest-PACS を DL の大規模計算環境として提供することを目的としている．そこで ChainerMN による性能評価を行なった．ChainerMN は GPU クラスターにおいて複数ノードでの実行において高い性能を示しているが，CPU 向けの実装においては性能についてあまり考慮されてこなかった．よってメニーコアプロセッサに適用するためにまずは性能の分析を行なった．

表 4　評価環境

Platform	Oakforest-PACS
Memory Mode	Cache
Dataset	ImageNet
Model	ResNet-50

表 5　使用ソフトウェア

Software	Version
Python	3.6.3
Intel MPI	2018.1.163
MPI4py	3.0.0
Chainer	5.0.0
iDeep4py	2.0.0

表 6　ImageNet データセット

Item	#
Training Data	1,281,167
Validation Data	50,000
Classification	1,000
Batch Size	32

4.1　評価環境

評価環境を表 4，使用したソフトウェアを表 5 に示す．Python は Intel 社が提供する Intel Distributed for Python[12] を使用した．これは従来の Python とは異なり，演算のコアになる Numpy や Scipy において Intel Xeon Phi 向けに最適化された MKL を使用できるため，性能の向上が期待できる．また，Chainer のバージョンは v5.0.0 を用いた．2018 年 4 月にリリースされた Chainer v4 から Intel Deep Learning Package(iDeep)[13] に対応し，Intel CPU での学習及び推論の高速化が実現されると共に，iDeep によって内部で OpenMP によるスレッド並列が可能となった．メモリモードは Cache に設定し，モデルは ResNet-50 を使用し，データセットは ImageNet データセットを用いた．データセットの詳細は表 6 に示す．また，ノードあたりのプロセス数は 1 とし，使用物理コア数を 64，スレッド数は 64 とする．これらの値は文献 [17] で行なった詳細な分析結果から最適値を設定した．

4.2　OpenMP+MPI によるハイブリッド並列

OpenMP によるマルチスレッド並列に加えて，MPI によるマルチプロセスを組み合わせたハイブリッド並列の結果を示す．これらの結果を含む詳細な分析は文献 [17] を参照されたい．図 5 は，理想的なスケーリングと実測した高速化率を示しており，128 ノードまではよくスケールしているが，256 ノードあたりから理想的なスケーリングからやや外れているのがわかる．図 6 は 1 イテレーションあたりの実行時間と内訳を示しており，図 5 と同様にノード数が増すごとに遅くなっていることがわかる．これらの結果はどちらも Collective 通信である All-Reduce の同期によるオーバーヘッドが原因であると考えられる．したがって，トレーニング時間を短縮するためには通信時間を減らす工夫が必要であると言える．

．図 6 においては 1 イテレーションあたりの速度であるため，計算時間は本質的に変わらないことから

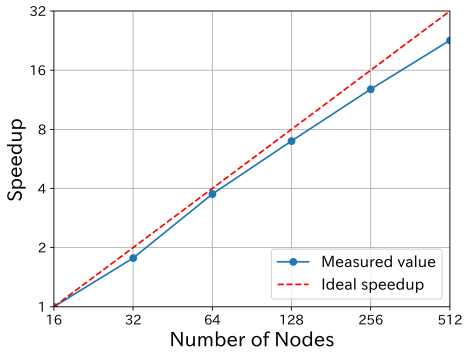

図 5　ノード数による性能向上

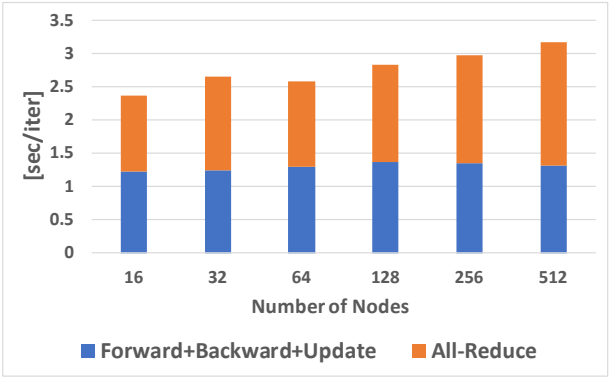

図 6　1 イテレーションあたりの実行時間と内訳

5.　設計・実装

5.1　従来手法

図 6 で示したように，トレーニング時間内における通信時間は，プロセス数の増加に伴い支配的になる．これは先述した通り，ChainerMN は CPU 向けの実装においては性能面のチューニングはあまりなされていないためである．したがって，通信においても図 7 に示すようなナイーブな実装がされている．まずは All-Reduce によって各プロセスが持つ勾配の和を求め，その後に各プロセスがプロセス数で割ることで勾配の平均を求めるという流れで処理が行われる

All-Reduce はメッセージサイズおよびコミュニケータ内の通信相手が増加するに伴ってオーバヘッドが増大する上に，メッセージサイズが大きいほど通信時間は急激に増加する，したがって，通信相手を複数のグループに分割し，通信相手を少なくすることで通信時間の短縮が可能であり，そしてメッセージサイズを分割することで，除算処理を伴う Average の処理も短縮ができると考えられる．

5.2　提案手法

本稿では三上ら [9] によって提案された 2D-Torus All-Reduce に対して One-Sided 通信を組み込むことで高速化を図った．まずは 2D-Torus All-Reduce について説明する．

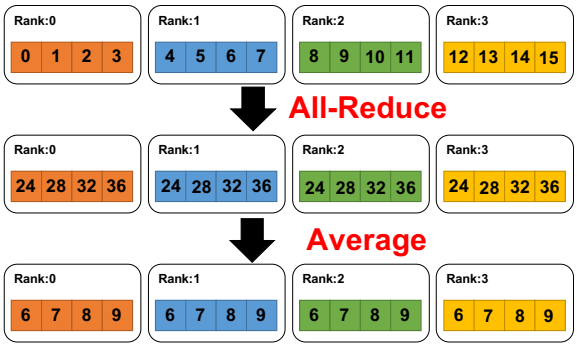

図 7　従来の通信手法

2D-Torus All-Reduce のアルゴリズムを図 8 に示す．まず各プロセスをグリッド状に配置し，水平方向に Reduce-Scatter, 垂直方向に All-Reduce, 水平方向に All-Gather と垂直方向の 3 ステップで通信を行う．プロセス数を P，水平方向のプロセス数を M，垂直方向の数を N とすると例えば Ring All-Reduce を用いた場合は $2(P-1)$ 回の通信が必要である一方で，2D-Torus All-Reduce は $2(M-1)$ 回まで抑えることができる上に，垂直方向の All-Reduce においては $1/M$ の短いメッセージサイズでの通信が可能になる．

All-Reduce や All-Gather と言った Collective 通信は同期によるオーバーヘッドが生じてしまう．そこで One-Sided 通信による通信の高速化を提案する．One-Sided 通信とは，通信相手の状態に関係なく，他のプロセスのデータをアクセスする通信方法である．このように他のノードにおけるメモリに対してアクセスする技術は Remote Memory Access(RMA) と呼ばれる技術であり，分散メモリ型を想定している MPI 上で，メモリを共有しているようなデータの操作が可能になる．これにより同期コストの削減が可能になるだけでなく，MPI 用に確保されたメモリ領域を経由する必要がないためデータコピーの削減による高速化も図ることができる．One-Sided 通信はこれらの利点を持ち合わせていながらも実装された例が少ない．したがって，本稿では One-Sided 通信における通信時間の評価を行う．

本稿では三上ら [9] によって提案された 2D-Torus All-Reduce に対して One-Sided 通信を組み込むことで高速化を図る．2D-Torus All-Reduce は Reduce-Scatter，All-Reduce，All-Gather の 3 ステップで構成されていたが，演算処理を含まない All-Gather を One-Sided 通信によって実現した．水平方向に並んだ 4 プロセスにおける One-Sided 通信の例を図 9 に示す．

まず，全てのプロセスがアクセス可能となる Window Object を生成する．次にプロセス同士で 2 組のペアを構築する．ペアは Recursive Doubling アルゴリズム [14] に基づいて選択をする．i 番目におけるペアの相手は $R \oplus 2^i$ によって得られる．そして，ペア同士でお互いがもつ勾配情報を自身の Window Object から相手の Window Object に Put する．この時，Put するメモリ先頭の先頭アドレスを指定することでメモリのコピーを避けることができる．これを繰り返すことで All-Gather と同様の結果が得られる．また，同期に関しては相手のペアからデータを得たタイミングで行う．全ての勾配を得るまで繰り返す回数はプロセス数を P とすると $\log_2 P$ 回で済む．

6. 評価

本研究は ChainerMN をメニーコアプロセッサへ適用し，OFP を DL 大規模計算環境として提供することを目指す．そこで 5.2 節で述べた提案手法によって通信時間の削減を実現することにより，その有用性を確認する．また，評価環境は 4.1 節に従うものとする．

6.1 One-Sided 通信による通信時間の評価

本稿では 2D-Torus All-Reduce における All-Gather を One-Sided 通信によって実現した．この時のイテレーションあたりの通信時間を示す．プロセス数は 256 で固定した．水平方向のプロセス数を M，垂直方向のプロセス数を N として（M, N）と表記した．図中の RSAA は ReduceScatter-AllReduce-AllGather の略で 2D-Torus All-Reduce を意味する．RSAP は ReduceScatter-AllReduce-Put の略で，All-Gather の代わりに One-Sided 通信の Put 関数を用いて実装した場合を表している．

RSAP においては，水平方向のプロセス数が多くなるにつれて通信時間が長くなっている．これは 5.2 節で述べたように水平方向のプロセス数を M とすると Put 処理を $\log_2(M)$ 回することになるため，その分処理に時間がかかってしまったためであると考えられる．一方で M を小さくすると All-Reduce を行うメッセージサイズが大きくなってしまうが、全体の通信時間としては RSAP の（2, 128）が最も高速であり，All-Reduce のみで実装した場合と比べて約 2.1 倍の高速化を達成しているため，One-Sided 通信による効果が大きいことがわかる．

6.2 All-Gather と Put の通信時間の比較

次に Put 関数と All-Gather 関数の通信時間の比較を行なった．結果を図 11 に示す．All-Gather に関しては，Put と比べると比較的小さいメッセージサイズにおいて有効であることがわかるが，メッセージサイズが 1[MByte] あたりになると Put よりも遅くなりはじめる．これは，一般に，One-Sided 通信は同期コストが少ないことやデータコピーを必要としないといった性質上，小さなサイズのメッセージを単独で送るような通信パターンには向いておらず，小サイズのメッセージを多く送るか，あるいは大きなメッセージを送るといった通信パターンにおいて効果を発揮できるためである．したがって，1[MByte] 以下程度のメッセージサイズにおいては All-Gather が有効ではある

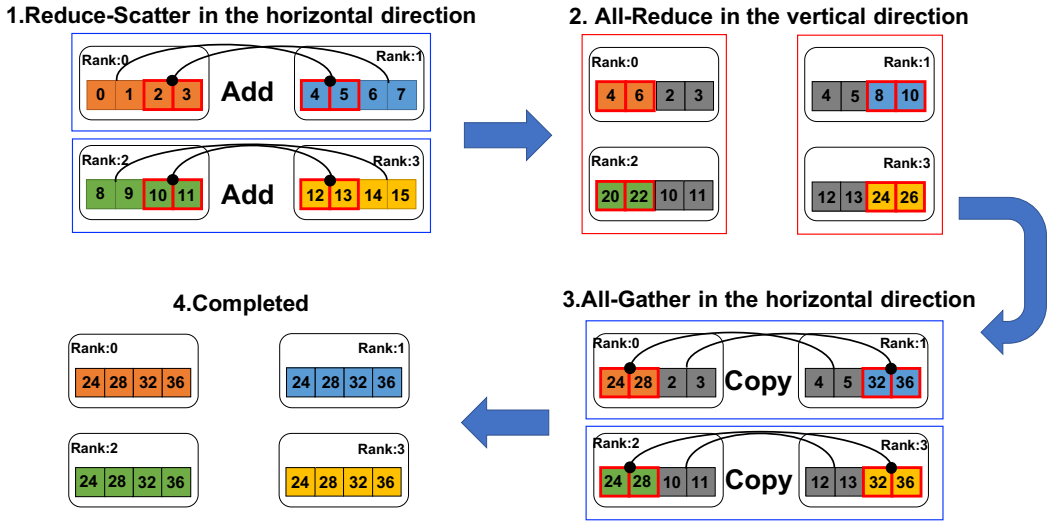

図 8　2D-Torus All-reduce を用いた実装

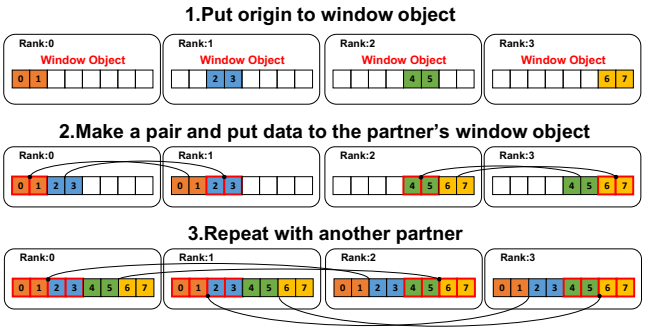

図 9　One-Sided 通信による All-Gather 実装

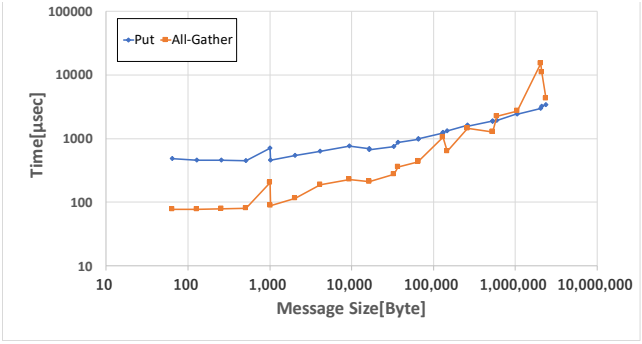

図 11　One-Sided 通信による All-Gather 実装

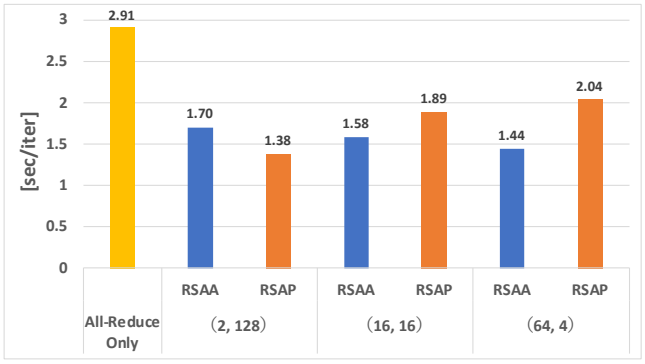

図 10　One-Sided 通信による All-Gather 実装

が，それを超えるメッセージサイズにおいては Put の方が高い性能が得られる．また，今回は同期に Fence を用いたが，同期のアルゴリズムによっても大きく性能が左右されるため，例えば PSCW により細かな同期の制御をすることで，さらなる高速化が期待できる

7.　まとめ

本研究では，分散並列学習フレームワークである ChainerMN を、メニーコアプロセッサ Intel Xeon Phi を搭載した Oakforest-PACS に適用し，性能評価および通信性能に関する高速化を実施した．まずは予備評価と分析において，OpenMP および MPI による並列化における評価を行った．1 イテレーションあたりの通信時間を計測することで，トレーニング実行時間において勾配の集約処理による通信が大部分を占めていることを明らかにした．そこで，三上ら [9] が提案した 2D-Torus All-Reduce を基に，All-Gather を One-Sided 通信によって実現し，同期コストやデータコピーの削減によって高速化を図った．その結果，従来の通信と比べて約 2.1 倍の高速化を達成した．この結果より，実行例が少ないメニーコアプロセッサを用いた DL の演算に関しても十分に高い性能が得られたことを示した．

また，本研究では，ImageNet/ResNet-50 で評価を行なったが，比較的サイズが小さいメッセージを多く通信するモデルであることから One-Sided 通信による大きな効果は得られなかった．したがって，seq2seq と呼ばれる RNN を用いた Encoder-Decoder モデルやその他のベンチマークにおける評価を実施することでその有用性を確認する．また，本稿においては 2D-Torus All-Reduce における All-Gather を Put 関数によって高速化を実現したが，Reduce-Scatter も Accumulate 関数によって同様の機能を実現できるため，可能な限り Collective 通信を使わずに One-Sided 通

信を使うことで更なる高速化を図る予定である．また，本稿においては精度に関する議論を含んでいないため，ImageNet/ResNet-50 ベンチマークによる精度の評価も行う必要がある．

謝辞 本研究について多くのご助言をいただいた Preferred Networks 社福田圭祐氏，鈴木脩司氏に深く感謝いたします．そして本研究についてご議論いただいた，東京大学情報理工学研究科 田浦健次朗教授，樋口兼一氏に感謝いたします．

参考文献

[1] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton: “Chainer: a next-generation open source framework for deep learning”, NIPS, 2015
[2] T. Akiba, K. Fukuda, and S. Suzuki: ”ChainerMN: scalable distributed deep learning framework”, CoRR, abs/1710.11351, 2017
[3] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang: ”On large-batch training for deep learning: Gen- eralization gap and sharp minima”, ICLR, 2017
[4] K. He, X. Zhang, S. Ren and J. Sun.: ”Deep Residual Learning for Image Recognition”, CVPR, 2016.
[5] Goyal, P., Dollár, P., Girshick, R., et al: ”Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”, arXiv preprint arXiv:1706.02677, 2017
[6] Samuel L Smith, Pieter-Jan Kindermans, and Quoc V Le: ”Don’t Decay the Learning Rate, Increase the Batch Size”, arXiv preprint arXiv:1711.00489, 2017
[7] T.Akiba, S.Suzuki, and K.Fukuda: ”Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes”, NIPS, 2017
[8] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, T. Chen, G. Hu, S. Shi and X. Chu.: ”Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes”, arXiv preprint arXiv:1807.11205, 2018
[9] Mikami, Hiroaki, Hisahiro Suganuma et al: “ImageNet/ResNet-50 Training in 224 Seconds”, arXiv preprint arXiv:1811.05233, 2018
[10] C. Ying, S. Kumar, D. Chen, T. Wang, and Y. Cheng.: ”Image Classification at Supercomputer Scale”, arXiv:1811.06992 [cs, stat], 2018
[11] A. Devarakonda, M. Naumov and M. Garland.: ”AdaBatch: Adaptive Batch Sizes for Training Deep Neural Networks”., arXiv preprint arXiv:1712.02029, 2017.
[12] “Intel distribution for Python”, https://github.com/IntelPython
[13] “Intel Deep Learning Package”, https://github.com/intel/ideep
[14] P. Kogge and H. Stone.: ”A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations”,. IEEE Trans. Computers C-22, 786–793, 1973
[15] T. Kurth, J. Zhang, N. Satish, et al.: ”Supervised and Semi-Supervised Classification for Scientific Data”, SC, 2017
[16] He Ma,Fei Mao,and Graham W Taylor.: Theano-mpi: a Theano-based Distributed Training Frame-work. arXiv preprint arXiv:1605.08325, 2016
[17] 田村紘平, 塙敏博: ”大規模メニーコアシステム Oakforest-PACS におけるディープラーニングの性能評価”, 情報処理学会研究報告, Vol.2018-HPC-165, 2018