Vol.2016-HPC-155 No.6
2016/8/8

MPI を用いた Deep Learning 処理高速化の提案

山崎雅文[†1]　笠置明彦[†1]　田原司睦[†1]　中平直司[†1]

概要：Deep Neural Network（DNN）の研究・開発は，DNN 構成の検討や数多くのハイパーパラメタの調整に試行錯誤を重ねる必要があり，またその評価には大規模なデータを用いた膨大な演算が必要である．そのため，DNN の学習処理を多数の GPU を用いて高速に実行するニーズが高まっている．この論文では，多ノードで学習処理を実行する場合の並列処理に対する課題を検討し，MPI を用いた大規模 GPU クラスタ向けのデータ並列による学習処理を提案する．勾配情報のノード間集約処理アルゴリズムとして，二分木集約方式，全量バタフライ方式，分割バタフライ方式を評価した．提案手法では取り扱うデータサイズに応じて全量バタフライ方式と分割バタフライ方式を選択するハイブリッド方式を採用し，さらにノード間集約処理を学習処理と重複させることで高いスケーラビリティを実現する．評価実験では各ノードの処理量を変えない条件（week scale）において 256 ノード・256GPU 構成で学習の処理速度を 217 倍高速化させた．

キーワード：Deep Learning，学習処理，MPI，GPU クラスタ，高速化

Accelerating Deep Learning Framework with MPI

MASAFUMI YAMAZAKI[†1]　AKIHIKO KASAGI[†1]
TSUGUCHIKA TABARU[†1]　TADASHI NAKAHIRA[†1]

1. はじめに

近年，画像や音声認識をはじめとして人間の「認識し，考える」事を機械で行う技術として，Deep Learning（深層学習）が注目されている．この技術は，神経ニューロンを模した Deep Neural Network（DNN）に画像や音声，文章などを学習させることであり，画像認識の分野では既に人間を超える認識精度[1][2][3]が達成されている．今後，医療分野[4]や自動運転[5]といった様々な分野への応用が検討されており，さらには創作活動といった人間独自のものと思われてきた分野への挑戦[6][7]も始まっている．

現在 DNN の研究・開発には，非常に長い期間が必要となっている．これは，DNN 構造の決定や学習時のハイパーパラメタの最適化の方法に対する厳密な理論が未だに構築されておらず，これまでの経験やノウハウ[8]をもとに数多くの試行錯誤を重ねたうえで決定する必要があるからである．もう一つの理由は，学習処理に膨大な演算が必要だからである．世界的な画像識別コンペティションである ImageNet Large Scale Visual Recognition Challenge（ILSVRC）2012[9]において画期的な認識精度を達成した AlexNet[10]では，1〜2 週間かけて 128 万枚の画像を学習させた．最近では認識精度を上げるために，さらに規模が大きく層が深い DNN が提案[2][3]されており，評価に必要な計算量は増大している．

そのため DNN 学習処理に対する高速化のニーズは非常に高く，高い並列処理性能をもつアクセラレータの利用が効果的である．現在，アクセラレータとして Graphics Processing Unit（GPU）を用いることが多い．これは GPU が多くの演算器と広帯域メモリインタフェースを持つハードウェアであり，汎用の開発環境 CUDA[11]や，深層学習用ライブラリ cuDNN[12]が提供されているためである．

また，DNN 開発や利用のためのソフトウェア・フレームワークである Caffe[13]は，筐体内の複数の GPU を用いて学習を高速化させる事も可能である．しかし，通常筐体内に搭載できる GPU は最大でも 4〜8 個程度に限られるため，それ以上の GPU を利用するには，独自で Message Passing Interface（MPI）等を利用し，複数の筐体（ノード）で分散処理を行う必要がある．

最近では Caffe に MPI を取り入れ，複数のノードの GPU を用いて学習処理を高速化した事例も報告されている．例えば，2016 年 3 月に発表された Caffe-MPI[14]は，16 個の GPU を用いて GoogLeNet[15]の学習処理を 13 倍高速に処理している．同じ 3 月に発表された fireCaffe[16]は，ノード間での転送量を抑えるために AlexNet ではなく NiN[17]や GoogLeNet [15]を用いて，学習の処理速度で 128 個の GPU を用いて 53 倍の高速化を達成している．

本稿では，より汎用的な DNN としてパラメタサイズの大きな AlexNet を用い，多数のノードに分散した GPU を用いて高速化を実現する処理方法を提案する．評価実験では Caffe に対し，GPU での DNN 学習処理時間と CPU でのデータ通信処理時間を可能な限り重複させ，ノード間の通信を転送データサイズに応じて最適な Allreduce 手法を選択するハイブリッド方式の実装を行った．その結果，各ノードに 1 つの GPU を搭載した 256 ノードの環境において，217 倍の処理速度の高速化を達成した．

†1 (株)富士通研究所
Fujitsu Laboratories Ltd.

2 章では DNN 学習処理について説明したのち，3 章でその並列化手法とノード間集約処理アルゴリズムについて述べる．4 章では並列化に伴う通信・集約処理時間を学習処理に隠蔽する手法を紹介し，5 章で評価環境と評価方法を説明する．6 章で評価結果を示した上で考察を行い，7 章で本稿の成果をまとめる．

2. DNN 学習処理

本章では DNN における基本的な学習処理の流れを説明する．また，DNN の学習処理において重要な要素となるハイパーパラメタについて述べる．

2.1 学習プロセス

DNN は多数のニューロンからなる層が複数積み重なる構造になっている．これらの層の間で相互にデータを伝播させ，入力データの認識や学習を行う．

学習処理は，フォワード処理とバックワード処理，内部パラメタの更新処理からなっている．フォワード処理は認識処理とも呼ばれ，第一の層（ボトム層）から最終段の層（トップ層）に向けて処理を行う．ボトム層は入力データ D と重みパラメタ w を使用して演算を行い，データを出力する．その出力は次の層の入力となり，同様に重みパラメタ w を使用してより上位の層の入力となる情報を出力する．トップ層の出力は，入力データ D に対する認識結果となる．次にバックワード処理は，トップ層の出力と正解から誤差関数 E を求め，各層のパラメタの勾配情報 ∇E を算出する．この処理は，フォワード処理の向きと逆向き（トップ層からボトム層の方向）に処理が進む．バックワード処理がボトム層まで完了すると，勾配情報 ∇E から内部パラメタの更新量 Δw を計算し，w の更新を行う．この一連の処理を何度も繰り返すことで DNN の学習が行われる．

また，繰り返し行われる学習処理の状況を確認するため，定期的にバリデーションテストと呼ばれる処理を行う．バリデーションテストは，学習データとは異なる画像を用いてフォワード処理を行い，その認識結果の正解率もしくは損失関数の値で確認する．図 1 は上記処理の流れを示しており，上段は学習処理とバリデーションテストの反復処理を，下段は 1 サイクルの学習処理を示している．

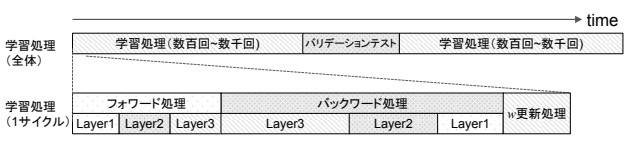

図 1　DNN の学習プロセス
Figure 1　Training process of DNN.

2.2 ハイパーパラメタ

DNN の学習処理には，調整が必要な多くのハイパーパラメタが存在[8]し，適切に設定することが必要である．本節では学習速度に関わる学習係数とバッチサイズについて述べる．

2.2.1 学習係数（Learning Rate）

学習係数とはバックワード処理により算出された勾配情報 ∇E を用いて重み更新量 Δw を算出する際の係数である．一般に学習係数が大き過ぎると重み w が発散してしまい，逆に小さすぎると収束までに時間がかかる場合や，局所極小解にトラップされ学習が止まる場合がある．

そこで，AlexNet では学習処理の初期の段階ではある程度大きな値の学習係数を設定しておき，段階的に減少させる手法を用いている．Caffe では，ハイパーパラメタとして，学習係数の初期値を base_lr，段階的に減少させる間隔を繰り返しのサイクル数で stepsize，減少させる際の係数を gamma として指定する．標準値は，base_lr は 0.01，gamma は 0.1 である．また，stepsize の標準値は 100,000 であるが，この値は 20 epochs（学習用画像 128 万枚の一度の処理が 1 epoch）である．

このほかにも，学習係数を連続的に変化させる方法[18]なども提案されている．

2.2.2 バッチサイズ

DNN の学習処理は，GPU が得意とする並列演算を効率よく実行するために，一度に複数の画像を処理する．一度に処理する画像枚数がバッチサイズであり，この値は GPU のメモリサイズによる制約から，一般的に数十から数百の値が選択されることが多い．

また，バッチサイズを大きくすると，平均化された ∇E による w の更新量 Δw が安定するために，学習係数を大きくとることも可能となるが，一般にバッチサイズを N 倍にしても学習係数を N 倍にはできないために，最終的な学習速度は下がるとの指摘もある．バッチサイズを大きくする場合でも，識別するクラス数と同程度までとすることが良い[8]とされている．

3. DNN 学習処理の並列化手法

本章では DNN 学習処理の並列化に関して，これまでに提案されているモデル並列とデータ並列を説明し，さらにデータ並列を行う場合に必須となる Allreduce 手法の一般的なアルゴリズムについて述べる．

3.1 モデル並列とデータ並列

DNN 学習処理を複数の演算器に分割する方式として，モデル並列とデータ並列が提案されている．

モデル並列は，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[19]であり，演算器間を各層の入出力データが転送される．一つの DNN を複数の演算器に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，一方で各演算器の処理量を均一にすることは困難である．さらに各フォワード処理や各バッ

クワード処理において，それぞれの演算器間で転送されるデータの要素数にばらつきが大きくなりやすく，バッチサイズにも比例して増大する．そのため，ノード数が増加するにつれて，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特徴がある．

データ並列は，各演算器が同じ DNN モデルを保持した状態で，それぞれ異なる学習データ（バッチデータ）を用いて学習を進める方法である．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，w の更新を行った上で次のバッチ処理を行う．モデル並列と異なり，各演算器が独立してフォワード処理からバックワード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することが出来る．一方で，データ並列は DNN の重みパラメタのサイズによって決まる ∇E を通信し，集約するため，重みパラメタサイズの大きい DNN の場合，集約処理の負荷が大きくなる．

この勾配情報を集約する方式として，各演算器間で直接交換する方法のほかに，パラメタサーバと呼ばれる集約を行うサーバを用意する方式がある[20][21]．この方式では，各演算器はパラメタサーバへ ∇E を送信し，パラメタサーバから更新された ∇E を受け取る．演算器が増えるにつれてパラメタサーバに通信が集中し，集約演算量も増大するため，大規模な構成での高速化は困難である．

3.2 Allreduce による集約手法

演算器を搭載した複数のノードからなる環境でデータ並列を行う場合，ノード間の勾配情報 ∇E の集約には，Allreduce 処理による要素ごとの総和を用いる．一般に Allreduce 処理は，複数のノードがそれぞれ保持するベクトルデータに対し要素ごとの集約演算（総和，最大値，最小値，論理演算等）を行い，その結果を全ノードで共有する処理である．ノード間の通信は 1 対 1 通信が基本となるため，Allreduce 処理は通常複数フェーズでのノード間通信の組み合わせで実現する．Allreduce 処理の代表的なアルゴリズムとしては，図 2 で示される 3 種類（二分木集約方式，全量バタフライ方式，分割バタフライ方式）がある[22][23]．最上段の 4 つの矩形は，集約の対象となるデータ（各ノードで算出された ∇E）を示しており，各行は Allreduce の処理フェーズを示している．矢印はノード間の通信を示しており，太枠で囲った領域がそのフェーズでの転送対象，網掛けの部分が集約演算を行う対象である．

3.2.1 二分木集約方式（Binary tree Reduce Broadcast : BRB）

BRB は一つのノードにすべてのデータを集約した後，その結果をすべてのノードに共有する方法である．

対象となる ∇E のデータサイズを M，ノード数を N とした場合，システム全体でのデータの通信量と演算量のオーダは最小の $\theta(M \cdot N)$ となり，矢印で示した通信の回数も少ない．しかし特定のノード（図 2 では node 0）に，通信と演算が偏る．

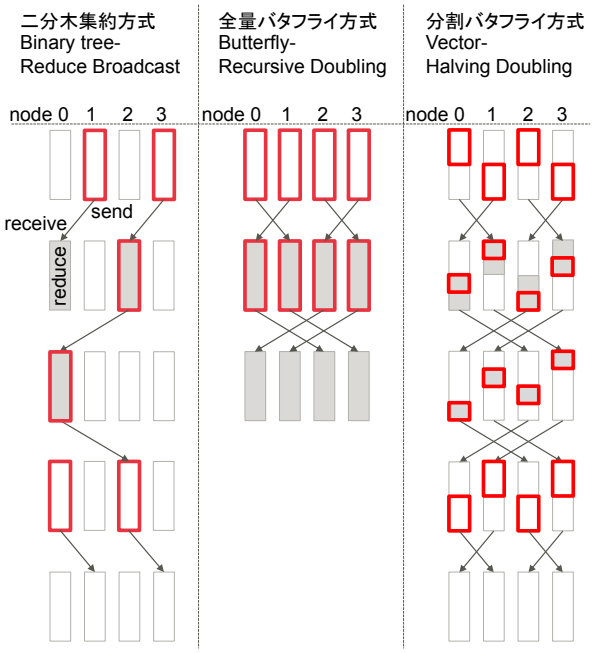

図 2　Allreduce 処理の 3 アルゴリズム
Figure 2　Three algorithms of Allreduce.

3.2.2 全量バタフライ方式（Butterfly Recursive Doubling : BRD）

BRD は，p フェーズ目にノード番号（rank 数）が $2^{(p-1)}$ だけ離れたノードとペアを作り，データを交換する方式である．

全ノードが同時に送信・受信を行うことで，最小のフェーズ数（$\log_2 N$）で Allreduce 処理が完了する方式である．ただし，各フェーズで全ノードが全データを交換し，集約処理を行うため，システム全体での通信量と演算量が大きくなる．ゆえに，集約演算処理を高速に実行できる場合，もしくは転送するデータサイズが小さい場合に有効なアルゴリズムといえる．

3.2.3 分割バタフライ方式（Vector Halving Doubling : VHD）

VHD は，BRD のように全体の通信量，演算量を増やさずに，BRB におけるノード間ばらつきを抑えたアルゴリズムで，ノードが各々 M / N 分の集約結果を持つようにノード間通信・集約処理を行い，その後すべてのノードで共有する方法である．図 2 に示すように，集約処理はフェーズごとに転送量が半分となり，その後に展開するときはフェーズごとに転送量が倍になるように行う．BRB のように特定のノードに通信・演算が集中することなく，BRD のように通信・演算が増えることはない．しかし，ノード数が多くなるに従い，フェーズ数が増えて転送するデータサイズが小さくなるため，要素数の少ないデータの Allreduce 処理を行う場合，ノード間通信におけるオーバヘッドにより，転送効率が低下する事もある．

以上 3 つの Allreduce アルゴリズムの通信量と演算量，処理フェーズ数をまとめ，表 1 に示す．

表 1　各 Allreduce 処理の通信量と演算量

Table 1　Amount of communication and computation of each allreduce algorithms.

Type	1 ノードあたり（最大）		全ノード合計		処理フェーズ数
	通信量	演算量	通信量	演算量	
BRB : Binary Reduce Broadcast	$2\times M\times\log_2 N$	$M\times\log_2 N$	$2\times M\times(N\text{-}1)$	$M\times(N-1)$	$2\times\log_2 N$
BRD : Butterfly Recursive Doubling	$M\times\log_2 N$	$M\times\log_2 N$	$M\times\log_2 N\times N$	$M\times\log_2 N\times N$	$\log_2 N$
VHD : Vector Halving Doubling	$2\times M$ 以下	M 以下	$2\times M\times(N\text{-}1)$	$M\times(N-1)$	$2\times\log_2 N$

4. DNN 学習処理の高速化

本章では, DNN 学習処理を並列に高速処理する方法を説明する．Allreduce 通信時間を GPU による学習処理時間に隠蔽する手法を中心に説明する．

ここでは，3 章で述べたように規模の大きな並列化を想定し，並列化手法としては，パラメタサーバ無しで各ノード間通信による ∇E 集約を行うデータ並列を用いる．また Allreduce 処理は，転送のデータサイズが小さい場合にはフェーズ数が最小の BRD，データサイズが大きい場合には VHD を切り替えて使用する．

4.1 バックワード処理と通信時間の並列処理

バックワード処理において勾配情報 ∇E は層ごとに算出される. そのため, 全ての層の勾配情報の算出を待たずに, 層単位で集約処理を開始することができる. このとき通信, 集約処理を CPU にて実行することにより GPU によるバックワード処理を妨げることなく処理が可能である．集約処理の手順は以下の通り．

Step 1: GPU メモリ領域の ∇E を CPU メモリ領域へコピー
Step 2: MPI を用いて ∇E の Allreduce を実行
Step 3: ∇E を GPU メモリ領域へコピーし，Δw を算出
Step 4: Δw を用いて w を更新

Step 2 の処理は GPU とは非同期に CPU 上で動作されるため，GPU の処理を中断させることなく，実行できる．図 3 は, GPU によるバックワード処理と w 更新処理(Step 4)，CPU による Step 1 から Step 3 の集約処理の流れを示しており，バックワードの演算処理と集約処理が並列に動作していることが分かる．

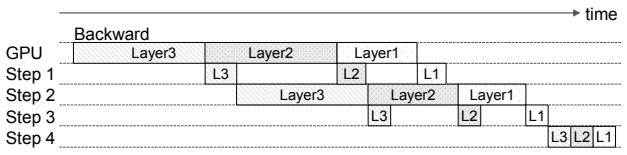

図 3　バックワード処理と通信時間の並列処理

Figure 3　Time chart of backward computation and allreduce processing.

4.2 転送処理の細分化

高速化のため，∇E の集約処理をさらに細分化して行う方法を提案する．Step 1 では，細分化された ∇E を GPU から CPU へコピーを行う．CPU へコピーが完了次第，Step 2 の通信処理を実行することが出来るため，細分化によって GPU-CPU 間の転送と集約処理を重複させる事が出来る．Step 2 では細分化されたデータに対して Allreduce を実行し，細分化されたデータ単位で全てのノードの ∇E を集約する．Step 2 での計算は，複数のスレッドを用いて並列に処理を行うことで，細分化された勾配情報ごとに非同期に通信・集約処理を実行することが出来る．

図 4 は一つの層（Layer n）の処理を，4 つに細分化し，Step 2 処理を 3 つのスレッド（Step 2-1, 2-2, 2-3）で行った場合の例を示している．細分化を行う事で，転送・集約処理が完了するまでの時間を削減する事が可能となっている．提案手法では，細分化するサイズやスレッド数を調整することで，CPU と GPU の計算資源を最大限利用し，高い処理速度を実現する．

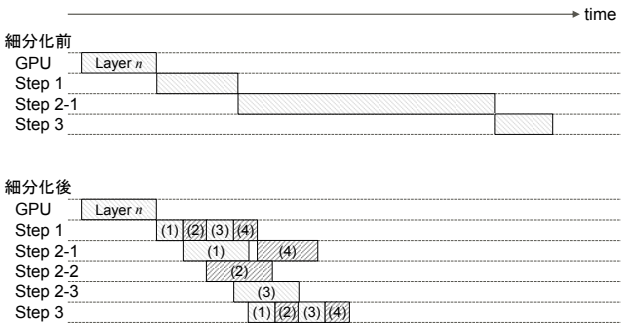

図 4　転送処理の細分化

Figure 4　Fragmentation of communication processing.

4.3 転送処理の追い抜き，フォワード処理の前倒し

GPU 側の処理時間が短い場合（ノード当たりのバッチ数が小さい場合など演算処理の負荷が軽い場合）, すべての集約処理が完了するまで，GPU は待機することになる．

4.2 節で述べた通り，勾配情報 ∇E の集約処理は，複数のスレッドで並列，非同期に実行されるため，∇E 要素数の小さな層のノード間転送や集約処理は，∇E 要素数の大きな層の処理よりも先に完了することが可能である．

ボトム層側から始まるフォワード処理を，それぞれの層の w 更新処理が完了した時点とすることで，より早い段階

でフォワード処理を開始できる．すなわち，学習処理のサイクルの中での通信時間を，バックワード処理時間に加え，フォワード処理の一部の時間も利用して隠蔽することが可能となる．

図 5 は，Layer ごとの allreduce 処理における，処理の追い抜きを示しており，Layer 1 の Forward 処理は，Layer 2 の完了を待たずに開始される．

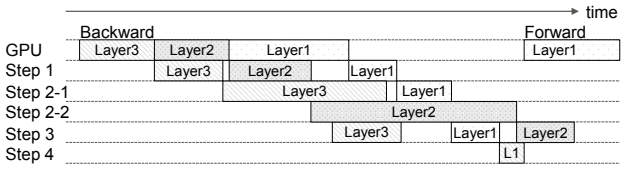

図 5　allreduce 処理の追い抜き
Figure 5　Overtaking of allreduce processing.

5. 評価環境と評価方法

4 章で述べた高速化手法を Caffe に実装し，評価を行った．5 章では，評価に用いた環境と評価方法について述べる．

5.1 評価環境

評価環境として，2 つのスーパーコンピュータ環境を使用した．1 つは九州大学の高性能演算サーバシステムで，1 つのノードあたり 1 つの GPU を搭載し，ノード間には，高速・低レイテンシの InfiniBand インターコネクトが採用されている．主に高速化手法の確認及び基礎データの取得を行った．

その後，東京工業大学の TSUBAME2.5 において，学習速度の実測を行った．本システムは，1 つのノードに 3 つの GPU を搭載した構成となっているが，今回の評価ではノード間の通信がボトルネックとなる状況で評価を行うため，GPU 1 つのみを使用した．こちらのシステムもノード間インターコネクトは InfiniBand である．

九州大学 情報基盤研究開発センター 高性能演算サーバ
ノード数: 1476 ノード（うち，最大 16 台を使用）
CPU: 16 core/node, 128 GB/node
GPU: Tesla K20m x1/node, 1.17 TFlops/GPU, 5 GB/GPU
Software : CUDA7.5, cuDNN v4.0, Intel MPI 4.0.3
InfiniBand FDR x1/node（6.8 GB/s）
東京工業大学 学術国際情報センター TSUBAME2.5
ノード数: およそ 1400 台（うち，最大 256 台を使用）
CPU: 12 core/node, 54 GB/node
GPU: Tesla K20X x3/node, 1.31 TFlops, 6 GB
Software : CUDA7.5, cuDNN v4.0, OpenMPI1.6.3
InfiniBand QDR x2/node（4.0 GB/s x 2）

評価には，AlexNet[10]を用いた．その層構成とデータ要素数，パラメタ要素数を表 2 に示す．表 2 から，入力層（ボトム層）に近い層はパラメタ数が少ないが，トップ側には，パラメタ数が非常に大きな層が存在していることがわかる．

画像データには，ILSVRC に用いられる ILSVRC データセットを用いた．学習用画像データ約 128 万枚，検証用の画像データは 5 万枚からなる画像データは予めノード数と同じ数に分割してあり，プログラムの開始時にノードごとのストレージへコピー（ステージング）した後に学習処理を実行した．これにより，学習用データの読み出しによるノード間通信への影響を回避している．

重みや勾配情報には 32 bit 単精度浮動小数を用いた．

表 2　AlexNet の層構成
Table 2　The layer structure of AlexNet.

層	種別	データ 要素数 |D| (*)	パラメタ 要素数 |W|
入力層	Input	40M	-
Conv1	畳み込み	74M	35K
Pool1	Pooling	18M	-
Conv2	畳み込み	48M	307K
Pool2	Pooling	11M	-
Conv3	畳み込み	17M	885K
Conv4	畳み込み	17M	664K
Conv5	畳み込み	11M	442K
Pool5	Pooling	2M	-
FC6	全結合	1M	38M
FC7	全結合	1M	17M
FC8	全結合	512K	4M
Softmax	Softmax	256K	-
Total		**240M**	**61M**

(*)バッチサイズが 256 の場合

5.2 評価方法

評価は 3 段階で行った．それぞれの目的，方法について以下に示す．

5.2.1 処理速度のスケーラビリティ

処理速度のスケーラビリティは，高速化率 s を用いて評価を行った．ノード数を N，ノード当たりのバッチサイズを b とし，フォワード処理からバックワード処理，w 更新までの学習処理 1 サイクルの時間を $t_{N,b}$ とする．基準として，1 個の GPU で，バッチサイズ 256 で行う場合の学習処理 1 サイクルの時間 $t_{1,256}$ とすると，学習処理 1 サイクル当たりの処理量は（$N \times b$ / 256）倍であるから，高速化率 s は次の式で表す事ができる．

$$s_{N,b} = \frac{t_{1,256}}{t_{N,b}} \times \frac{b}{256} \times N \qquad \cdots(1)$$

ノード間並列を採用した場合の学習の処理時間 $t_{N,b}$ を決定する要因としては大きく 2 つが考えられる．一つは主に GPU で行われる行列演算や畳み込み演算といった学習処

理に必要な演算量であり，これはノード当たりのバッチサイズにほぼ比例して多くなる．また，演算量は DNN の層構成によっても大きく変わり，一般的に全結合層では演算量は少なく畳み込み層は演算量が多い．

もう一つの要因は，ノード間転送に要する時間であり，バッチサイズには影響を受けないが，ノード数 n に依存する．ほかにも勾配情報 Δw のサイズ，ノード間通信の固定遅延（レイテンシ）にも依存する．

演算量が多い場合，集約処理時間はほぼ GPU 演算時間に隠蔽されノード数にかかわらずほぼ一定に近づくため，スケーラビリティは良くなると考えられるが，演算時間が短くなると，全体の処理時間に対する集約処理時間が増加するため，スケーラビリティは悪化すると考えられる．

5.2.2 DNN 学習へのバッチサイズの影響

各ノードの処理量を変えずに処理ノードを増やした場合，バッチサイズは大きくなる．このとき，学習の進み方がどのように変化するか，バリデーションテスト時の top 1 正解率を用いて評価を行った．

本評価実験では，学習の開始時に重みパラメタ w の初期値をすべてのノードで共有する．また，全てのノードで共通の乱数の種を使用することで，学習処理の中で使用される乱数をノード間で一致させた．これにより，ノード数 N で分割して実行した場合，ノード当たりのバッチサイズ b の N 倍の巨大なバッチサイズを実現した．

5.2.3 学習速度の高速化

最後に，1 サイクル当たりの学習処理速度とバッチサイズに影響を受ける学習の進み方の測定値から，ある認識精度（バリデーションテストにおいて，top 1 正解率が 40%, 45%, 50%）となるまでに必要な時間を算出し，1 ノードで学習を行った場合にかかる時間との比較を行った．

実際の DNN 学習処理においては，処理開始時のイメージデータ転送や，データ拡張を含む取り込み処理，また一定時間ごとのバリデーションテストなどの時間が加算されるが，今回は評価の対象外とした．

6. 結果と考察

6 章では，評価の結果について述べ，考察を行う．

6.1 処理速度のスケーラビリティ

ノード数 N と，ノード当たりのバッチサイズ b に対する，DNN 学習処理 1 サイクル当たりの処理時間 $t_{N,b}$ を測定し，算出した高速化率 s を図 6 に示す．図 6 において，縦軸は，1 ノード，b = 256 での処理速度に対する倍率であり，横軸はノード数である．点線は理想的な高速化率（N ノードで N 倍）を示している．

今回の測定では，256 ノードで 217 倍，64 ノードで 60.4 倍の処理速度の高速化を達成した．ただし，バッチサイズが小さくなるにつれて，高速化率は下がる．b = 256 から b = 64 とバッチサイズを 1/4 にすると，64 ノードで，30.5 倍，b = 32 とすると，15.5 倍まで下がってしまう．つまり b < 64 の領域では，式（1）において N をさらに k 倍に増やしても $t_{N\times k,\,b}$ / $t_{N,\,b}$ が 1 / k 程度となり，性能は変わらないことを示している．この結果から，AlexNet を使用する場合，ノード当たりのバッチサイズは 64 程度を確保することが望ましいことがわかる．

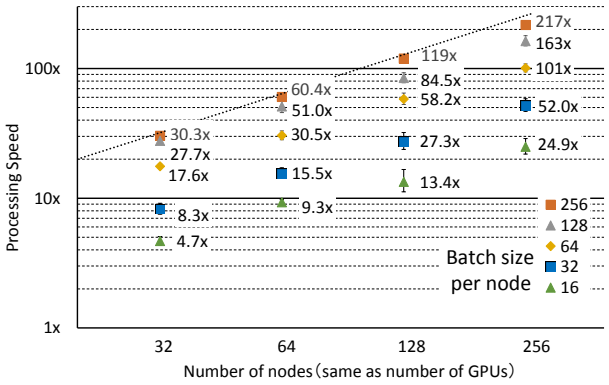

図 6　処理速度のノード数とノード当たりのバッチサイズ依存性

Figure 6　Dependence of processing speed on number of nodes and on batch size per node.

6.2 DNN 学習へのバッチサイズの影響

図 7 に，全体のバッチ数を増加させた場合の学習の進み方を示している．縦軸は，バリデーションテスト時の top 1 正解率，横軸はバッチ処理の処理回数を表しており，実時間ではない．本実験で用いたハイパーパラメタの値は，表 3 の通りである．

図 7 より，バッチサイズが大きくなるにつれ，少ない処理回数で top 1 正解率が上昇するが，50 epochs 実行時の最終正解率も下がっていることが分かる．この最終正解率については，学習係数の初期値（base_lr）やその切り替えタイミング（stepsize）をチューニングすることで同じ値に近づくことを確認している．

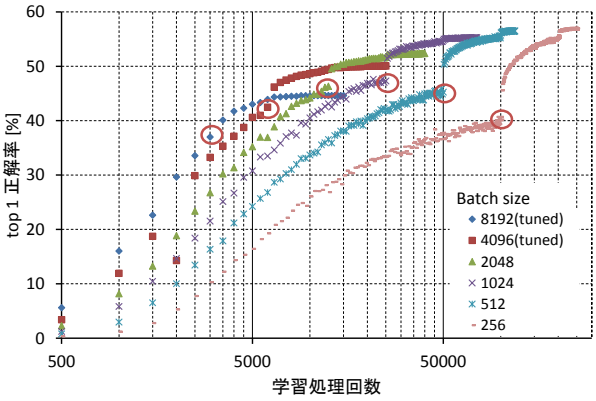

図 7　学習の進み方のバッチサイズ依存性

Figure 7　Dependence of training progress on batch size.

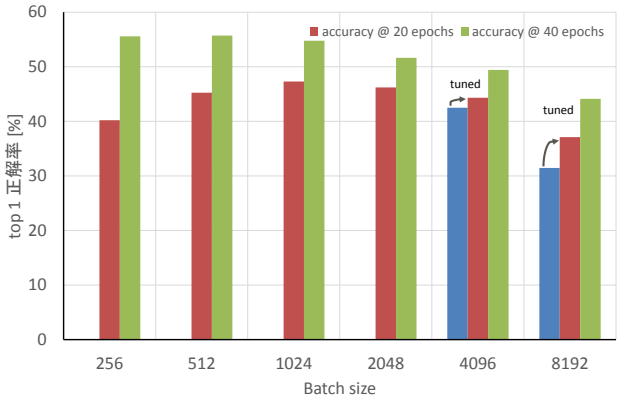

図 8　20, 40 epochs 時の top 1 正解率

Figure 8　Top 1 accuracy of each batch size on 20 epochs and 40 epochs.

表 3　実験で使用したハイパーパラメタ

Table 3　Hyper parameters on our experimental results.

Batch Size	base_lr	stepsize	gamma
256	0.01	100,000	0.1
512	0.01	50,000	0.1
1024	0.01	25,000	0.1
2048	0.01	12,500	0.1
4096	0.01	6,250	0.1
8192	0.01	3,125	0.1
4096 (tuned)	0.02	6,250	0.1
8192 (tuned)	0.03	3,125	0.1

また，図 7 において丸で示した点は，20 epochs 処理した時点を示しており，ここで learning rate が 0.1（gamma）倍に更新されている．バッチサイズに対する 20, 40 epochs 時の top 1 正解率を図 8 に示す．

図 8 より，20 epochs 経過時点で最も学習の進んでいるバッチサイズ 1024 は，識別クラス数 1000 と同程度であった．また，バッチサイズが 4096 では learning rate のチューニングにより認識精度は改善されている．

6.3 学習速度の高速化

表 4 は，6.1 節と 6.2 節の結果を用いて，複数のノードで AlexNet の学習処理を行った場合にかかる時間を算出した結果である．学習の判定条件は，バリデーションテスト時の top 1 正解率が 40%, 45%, 50%に到達するまでとし，さらに 1 台の GPU での実行時間に対する高速化の倍率も併記した．ノード当たりのバッチサイズ b は，6.1 節にて述べた最低値 64 とし，ノード数を 16 から 128 とした．

top 1 正解率が 40%に達するまでの学習時間（初期学習時間）としては，128 ノードで 54.7 倍の高速化を達成した．また，表 4 に下線で示した通り，8 ノードから 32 ノードにおいては実際の並列数よりも学習速度が向上している．これは，図 8 で示した通り学習回数が同じ場合に，より効率よく学習できるバッチサイズ（512 – 2048）であることによる．

また，top 1 正解率が 50%に達した時点でみてみると，16 ノードで 10.1 倍，32 ノードで 15.8 倍，64 ノードで 19.2 倍となった．128 ノード（b = 8192）の場合，50 epochs まで実行したものの accuracy は 50%に到達しなかった．

6.4 高速化に有効なノード数

DNN 学習のデータ並列による複数ノードで実行する場合の高速化について，スケーラビリティを確保するためには，ノード間集約に必要な通信と集約処理時間を，GPU 処理時間に対して隠蔽する必要がある．ノード間集約に必要な時間は，環境を構成するハードウェアと通信アルゴリズムに依存しており，GPU 処理時間は DNN の規模とバッチサイズにより決まる．今回評価に用いた環境と AlexNet の組み合わせの場合，ノード当たりのバッチサイズは約 64 以上を確保する必要があった．

また，DNN の学習はバッチサイズに大きく依存し，最適なサイズは識別クラス数と同程度であった．

よって，今回の条件においては，データ並列による高速化は，16 ノードで最大の効率となり，64 ノードまでは高速化に有効であるが，128 ノード以上では最終的な認識精度の悪化が顕著となった．

表 4　top 1 正解率が 40%, 45%, 50%に到達するまでの時間

Table 4　Train time to reach 40%, 45% and 50% of top 1 accuracy.

DNN	Batch size	Batch size of nodes	ノード数 (=GPUs)	accuracy=40% Train time	Speedup	accuracy=45% Train time	Speedup	accuracy=50% Train time	Speedup
AlexNet	256	256	1	23:34	1x	25:13	1x	27:42	1x
	512	64	8	2:06	11.1x	4:26	5.7x	5:28	5.1x
	1024	64	16	1:03	22.2x	1:43	14.7x	2:08	10.1x
	2048	64	32	0:33	42.0x	1:14	20.3x	1:45	15.8x
	4096	64	64	0:36	38.9x	0:56	27.0x	1:26	19.2x
	8192	64	128	0:25	54.7x	0:39	38.8x	-	-

7. おわりに

DNN の学習処理を GPU クラスタ環境で効率よく実行するために，ノード間通信・集約処理を GPU 等のアクセラレータでの処理と並列実行させ，通信・集約処理時間を隠蔽させる技術について述べた．隠蔽の可否は，ノード間集約に必要な通信と集約処理時間と，GPU 処理時間で決まる．ノード間集約に必要な時間は，環境を構成するハードウェアと通信アルゴリズム，ノード数に依存し，GPU 処理時間は DNN の構成とバッチサイズにより決まる．

処理速度のスケーラビリティは，相対的に DNN が大きくなるほど良好となる．同様に，パラメタサイズが大きく演算量が小さな全結合層よりも，パラメタサイズが小さく演算量が大きな畳み込み層が多い方が有利である．また，SqueezeNet[24]のように，認識精度を下げずにパラメタサイズを小さくするアプローチも有効である．

現在，理論的に DNN 構成やハイパーパラメタを一意に決めることができないため，最適化には繰り返しの実験が必要である．GPU クラスタ環境に本手法を用いることで，DNN 開発の大幅な時間の短縮が可能である．

GPU クラスタ環境の利用により，さらに高精度な DNN の開発・さまざまなアプリケーションへの応用が広がることを期待している．

謝辞 本研究は主に九州大学情報基盤研究開発センターの研究用計算機システム及び，東京工業大学学術国際情報センターの TSUBAME2.5 を用いて行った．また本研究の一部は HPCI システム利用研究課題の成果によるものである（課題番号:hp160240）．ここに感謝の意を表する．

参考文献

[1] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, IEEE International Conference on Computer Vision (ICCV), 2015
[2] C. Szegedy, V. Vanhoucke, S. Ioffe, and J. Shlens, “Rethinking the Inception Architecture for Computer Vision”, In Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.
[3] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition”, arXiv:1512.03385, 2015
[4] AA Cruz-Roa, JEA Ovalle, A. Madabhushi, and FAG Osorio, “A Deep Learning Architecture for Image Representation, Visual Interpretability and Automated Basal-Cell Carcinoma Cancer Detection”, Medical Image Computing and Computer-Assisted Intervention 2013, 2013
[5] C. Chen, A. Seff, A. Kornhauser, and J. Xiao. “DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving”, In Proceedings of 15th IEEE International Conference on Computer Vision (ICCV), 2015
[6] LA Gatys, AS Ecker, and M. Bethge, “Image Style Transfer Using Convolutional Neural Networks”, The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016
[7] 松原仁, 佐藤理史, “星新一に学ぶショートショートの自動創作”, The Japanese Society for Artificial Intelligence (JSAI), 2014
[8] G. Hinton, “A Practical Guide to Training Restricted Boltzmann Machines”, UTML TR 2010-003, 2010
[9] O. Russakovsky, J. Deng, H. Su, et al., “ImageNet Large Scale Visual Recognition Challenge”, International journal of Computer Vision (IJCV), 2015.
[10] A. Krizhevsky, I. Sutskever, and GE Hinton, “ImageNet Classification with Deep Convolutional Neural Networks”, Advances in Neural Information Processing Systems (NIPS), 2012
[11] nVIDIA, “CUDA C Programing Guide v7.5”, 2015
[12] nVIDIA, “cuDNN User Guide v4.0”, 2016
[13] Berkeley Vision and Learning Center (BVLC), Caffe, http://caffe.berkeleyvision.org/, 2013
[14] inspur, “Caffe-MPI”, https://github.com/Caffe-MPI/, 2016
[15] C. Szegedy, W. Liu, Y. Jia, et al, “Going deeper with convolutions”, IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015
[16] FN Iandola, K. Ashraf, MW Moskewicz, and K. Keutzer, “FireCaffe: Near-Linear Acceleration of Deep Neural Network Training on Compute Clusters” , arXiv:1511.00175, 2015.
[17] M. Lin, Q. Chen, and S. Yan, “Network In Network” , International Conference on Learning Representations (ICLR), 2014
[18] J. Duchi, E. Hazan, Y. Singer, “Adaptive Subgradient Methods for online learning and stochastic optimization”, Journal of Machine Learning Research, 2011
[19] A. Coates, B. Huval, T. Wang, et al., “Deep learning with COTS HPC systems”, In Proceedings of the 30th International Conference on Machine Learning (ICML), 2013
[20] J. Dean, GS Corrado, R. Monga, et al., “Large Scale Distributed Deep Networks”, Neural Information Processing Systems (NIPS), 2012.
[21] R. Wu, S. Yan, Y. Shan, et al., “Deep Image: Scaling up Image Recognition”, arXiv:1501.02876, 2015.
[22] 松本幸, 安達知也, 田中稔ら, “MPI_Allreduce の「京」上での実装と評価”, 情報処理学会研究報告, 2011
[23] R. Rabenseifner, “Optimization of Collective Reduction Operations”, In Proceedings of the International Conference on Computational Science (ICCS), 2004
[24] FN Iandola, S. Han, MW Moskewicz, et al, “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size”, arXiv:1602.07360, 2016