Vol.2019-HPC-170 No.9
2019/7/24

大規模機械学習訓練におけるI/O性能の高速化

芹沢 和洋[1,a)]　建部 修見[2]

概要：深層ニューラルネットワークに使用される訓練データの規模は年々増加しており，深層ニューラルネットワークの訓練処理において訓練データの read I/O は無視できないボトルネックになりつつある．ノードローカルストレージとして搭載された SSD などの I/O が高速なストレージデバイスを使用することで read I/O の高速化は可能であるが，HPC クラスタにおいては毎回訓練データセットのファイルコピーが毎回必要であるという課題がある．また，HPC クラスタの計算ノードからネットワークを経由してアクセス可能な外部ストレージは訓練データセットをファイルコピーせずに訓練処理を開始できるが，SSD ほどのバンド幅は見込めない．本研究では，ノードローカルストレージと外部ストレージを組み合わせて使用することで事前に訓練データセットのコピーをせずに read I/O を高速化する手法を提案する．提案手法を機械学習フレームワークである Chainer に実装し，Chainer が提供する並列に訓練データを read する機能をベースラインとして，read I/O 性能を自作したベンチマークによって比較したところ，Lustre に訓練データを配置した場合のベースラインよりも，より少ないプロセス数を使用して最大で約 20%高い read I/O 性能を達成できることを示した．データ並列訓練における 10 epoch の訓練時間の比較では，訓練データセットのファイルコピーに要する時間を考慮するとベースラインと SSD の組み合わせよりも訓練処理時間を短縮できることを示した．一方で，データ並列訓練においては read I/O ではなく AllReduce による処理時間が律速するため，ストレージ間の I/O 性能が処理時間に反映されにくいという，データ並列訓練の所要時間における特性を明らかにした．

キーワード：深層ニューラルネットワーク，Chainer，ノードローカルストレージ

1. はじめに

深層ニューラルネットワークで使用される訓練データセットの規模は年々増加している．例えば，Imagenet Large Scale Visual Recognition Challenge 2012 において，約 128 万枚の訓練画像データセット [1] が使用され，YouTube-8M Kaggle challenge においては 1.71 TB のフレーム画像データセットが使用された [2]．また，合計 3.5TB の気象シミュレーションのアウトプットを深層ニューラルネットワークで訓練した例も報告されている [3].

このような大規模の訓練データを用いる場合，深層ニューラルネットワークの訓練処理における訓練データの read I/O の影響が訓練処理時間に与える影響が無視できなくなる．例えば，畳み込みニューラルネットワークの 1 つである VGG16[4] を ImageNet を用いて訓練した際の処理時間の内訳を調査したところ，訓練処理を高速化する上で訓練データの read がボトルネックになっているという評価結果が報告されている [5]．深層ニューラルネットワークの訓練処理の大半は行列演算のため，GPU などのアクセラレータを用いて高速化する手法が一般に採用される．しかし，訓練データの read がボトルネックとなっている状況下では，アクセラレータの性能を発揮させ高速化の恩恵を受けることが困難である．

訓練データの read I/O は，一般に NVMe SSD などの高いバンド幅を持つストレージデバイスを用いることで改善が可能である．例えば，日本国内で運用されている HPC クラスタである AI Bridging Cloud Infrastructure (ABCI)，TSUBAME，Cygnus では，ユーザが計算ノードに搭載されている SSD をノードローカルストレージとして利用することができる．しかし，これらのノードローカルストレージは，計算ジョブごとの一時的な領域として提供されているため，訓練処理を行う度に計算ノード上のデバイスに訓練データをコピーする必要がある．そのため，訓練データセットのコピーに要する時間が毎回必要となる．一方で，これらの HPC クラスタにおいては，一般に計算ノードから直接アクセス可能な外部ストレージが提供されている．これらの外部ストレージに訓練データを配置する場合に

[1] 筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻
[2] 筑波大学計算科学研究センター
[a)] serizawa@hpcs.cs.tsukuba.ac.jp

は，訓練データのコピーは不要となる．しかし，外部ストレージを構成する分散ファイルシステムのバンド幅は，一般的にノードローカルストレージには及ばない．

本論文では，このような特性をもつノードローカルストレージと外部ストレージの両方を組み合わせて，訓練処理における訓練データの read I/O に要する時間を削減する手法を提案する．そして提案手法を機械学習フレームワークである Chainer[6] に実装し，Chainer の既存機能を用いた場合との訓練時間の比較を行う．

2. 研究の背景と動機

本章では，まず深層ニューラルネットワークにおける処理内容の概要を説明する．続いて，本論文で対象とする深層ニューラルネットワークの訓練処理時における一般的な read I/O の特性について述べる．最後に，訓練処理を高速化するために使用される「データ並列訓練」における read I/O の特性について述べる．

2.1 深層ニューラルネットワークの訓練処理の概要

深層ニューラルネットワークの訓練処理では，ストレージから読み込んだ訓練データをニューラルネットワークに適用し，各レイヤにおける演算を経て出力が伝搬し，最後に損失関数により損失が計算される．このフェーズは一般に forward と呼称される．そして，求めた損失を最小化するようにネットワークのパラメータを更新する．このフェーズは一般に backward と呼称される．この一連の処理が訓練処理の 1 iteration であり，これを訓練データの数だけ反復することで，ニューラルネットワークのパラメータが最適化されていく．

また，この訓練処理の際，一般的にミニバッチ訓練という手法が使用される．これは一度に複数の訓練データをバッチとして 1 回の入力として使用する手法である．1 つのミニバッチに含める訓練データの個数は，一般に 16，32 等が使用される．また，一般的に，iteration ごとに 1 つのミニバッチ内における訓練データの組み合わせは毎回変更されるため，バッチを編成するために毎回異なる訓練データがストレージから read される．

2.2 深層ニューラルネットワークの訓練処理における read I/O

read I/O の観点で見ると，深層ニューラルネットワークの訓練処理中は，小規模なデータの random read が大量に実行されている．例えばミニバッチのサイズが 32 であれば，各 iteration ごとに random に選択された 32 個の訓練データが read されている．このようなパターンの random read は，一般に sequential read に比べて高いバンド幅を出すことが難しいとされている．そのため，機械学習フレームワークである Chainer および TensorFlow[7] では，訓練データを並列に read して read I/O のスループットを向上させる機能，及びバッチ単位でのメモリへの訓練データのバッファリングを非同期に行う機能を，それぞれ提供している．しかし，これらの機能を使用したとしても，ニューラルネットワークへのミニバッチの供給スピードが forward/backward の処理速度を下回る状況になると，ストレージデバイスの論理的なレイテンシやバンド幅が訓練データを供給するスピードを律速することになる．そのため，後述するように，訓練データを Lustre などの外部ストレージに配置する場合と，SSD などのノードローカルストレージに配置する場合とでは，同じ並列度で read してもその read I/O 性能には差が生じる状況が生じる可能性がある．

2.3 データ並列訓練における read I/O

訓練処理の高速化のための別のアプローチとして，複数の計算ノードを使用し並列に訓練を行う「分散訓練」と呼ばれる手法が存在する．分散訓練には大きく分けて 2 種類のアプローチが存在する [8]．

1 つ目のアプローチはモデル並列と呼ばれる，これは，深層ニューラルネットワークの機械学習モデルを分割して，複数のプロセスに訓練処理を割り当て，複数のプロセス全体で 1 つの機械学習モデルを訓練する手法である．複数プロセスで訓練処理を行うことで，1 iteration 当たりの訓練処理時間が短縮でき，また 1 台の計算ノードではメモリに乗り切らない巨大な機械学習モデルの訓練処理が可能となる．

もう 1 つのアプローチはデータ並列と呼ばれる．これは，訓練データセットをプロセス数に応じて分割し，複数のプロセスで分割された訓練データセットを用いて並列に訓練を行う手法である．複数のプロセスを用いて擬似的にミニバッチサイズを増加させることで訓練処理における 1 epoch 当たりの iteration 回数を削減し，訓練処理のスループットを向上させることが可能となる．一方で，各プロセスで保持している勾配の値を 1 iteration ごとに全プロセス間で平均を取って同期する必要があるため，1 iteration 毎に AllReduce による集団通信が発生する．データ並列訓練を用いて訓練速度を改善させた実例として，画像データセットである ImageNet[9] の 1,000 クラス分類を行う機械学習モデルの訓練処理を高速化した事例が報告されている [10] [11] [12] .

本論文では，分散訓練で頻繁に採用されるデータ並列訓練に着目する．データ並列訓練では，複数のプロセスが並列に共通の訓練データセットを read する．また，各プロセスにそれぞれ個別の GPU を割り当てて演算を高速化するために，複数の計算ノードにまたがってプロセスが配置されることが要求される．この 2 つの要件を満たすために，訓練データセットのストレージへの配置方法は次の 2 種類

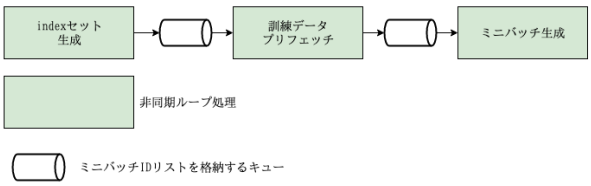

図 1 提案手法における処理フロー．各非同期ループ処理は前のステージのキューから index リストを pop し，処理が完了すると次のキューにショリ済みの index リストを push する．

が考えられる．

1 つ目は，計算ノード上のノードローカルストレージに配置する方法である．例えば，日本国内で運用されている HPC クラスタである AI Bridging Cloud Infrastructure (ABCI)，TSUBAME，Cygnus では，ユーザが計算ノードに搭載されている SSD をノードローカルストレージとして利用することができる．ノードローカルストレージに訓練データを配置することで，各プロセスが SSD の高いバンド幅を活かし高速に訓練データを read することができるメリットがある．一方で，上記の HPC クラスタでは，計算ノード上の SSD は計算ジョブ固有の一時領域として提供されるため，訓練処理を行う度に訓練データを毎回コピーしなければならないというデメリットがある．

2 つ目は，各計算ノードが共有する外部ストレージ上に配置する方法である．ABCI では GPFS，TSUBANE および Cygnus では Lustre によって構築された分散ファイルシステムが外部ストレージとして提供されており，各計算ノードから同一の訓練データセットにアクセスすることが可能である．そのため，訓練処理を行う度に訓練データセットをコピーする時間を省略することができるメリットがある．また，容量の面でも計算ノード上のノードローカルストレージよりも有利であるため，ノードローカルストレージの容量を超える訓練データセットを配置することも可能である．一方で，これらの分散ファイルシステムの read I/O 性能は一般にノードローカルストレージとして使用される SSD 等のデバイスよりも劣るため，read I/O 性能の面で不利になるというデメリットがある．

本論文では，この 2 種類の訓練データの配置方法を相補的に組み合わせることで，データ並列訓練における read I/O に要する処理時間を短縮させる方法を提案する．

3. 提案手法

本論文の提案手法における主要なアイデアは，外部ストレージからノードローカルストレージへの訓練データを並列に行うことで，外部ストレージからのファイルコピーに要する時間を短縮させる点である．具体的には，訓練データのコピーと，ノードローカルストレージからの訓練データの read を分離し，それぞれを独立した非同期並列処理

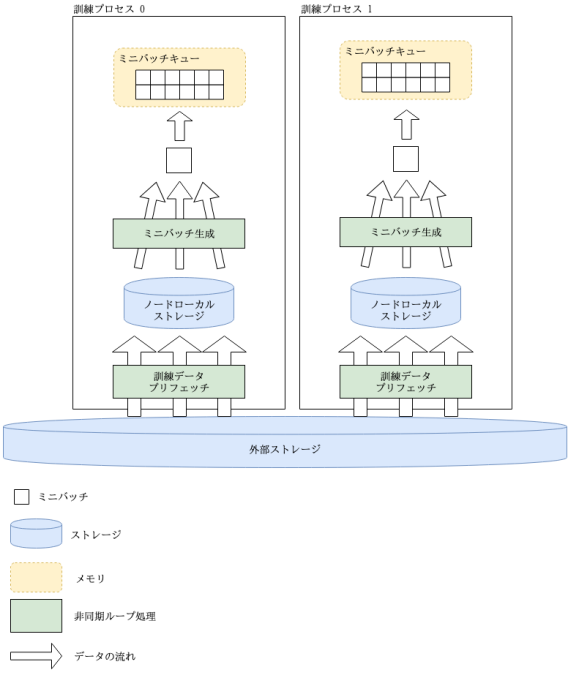

図 2 提案手法における訓練データの流れ

として実装する．また，ノードローカルストレージにすでにファイルがコピー済みの場合は，外部ストレージからのファイルコピー自体を省略することができる．これにより，訓練処理の開始時点ではノードローカルストレージにファイルが一切存在しない状態であるが，訓練処理が進むにつれて徐々にノードローカルストレージに訓練データがコピーされ，ノードローカルストレージに訓練データを配置して訓練する場合と同じような高速化の効果が得られる．

これを実現するために，提案手法では訓練データの input 処理を，一連の非同期並列パイプライン処理として実装する．このパイプラインは 3 ステージの非同期のループ処理から構成される．**図 1** に処理フローを示す．

まず，1 番目のステージで後続のステージで処理する訓練データを指定するための index リストをミニバッチ単位でを生成し，1 つ目のキューに push する．例えば，ミニバッチサイズが 32 であれば，32 個の index が含まれるリストが生成される．次に，2 番目のステージで index リストを 1 つ目のキューから pop し，含まれる index の訓練データを外部ストレージからノードローカルストレージへコピーする．その後，処理済みの index リストを 2 つ目のキューに push する．最後に，3 番目のステージで 2 つ目のキューから index リストを pop し，pop した index リストに対応する訓練データをノードローカルストレージからそれぞれ read し，ミニバッチを編成してメモリ上のバッファに格納する．この 3 ステージの処理はすべて非同期で実行され，訓練処理が終わるか，キューの上限サイズに達するまでループし続ける．

提案手法の実装には Chainer[*1]を使用した．Chainer を使用した理由は，コードの大部分が Python で実装されており[*2]，他の機械学習フレームワークと比べてコード全体の仕様理解および機能拡張に要するコストが低いと判断したためである．使用したコードは GitHub の chainer リポジトリの master ブランチから fork したものであり，fork 時点のバージョンは 6.0.0*rc*1 である．以下，それぞれのステージの詳細について説明する．

3.1 index リスト生成

まず，各 epoch の初めに訓練データセットの長さ分のランダムな整数リストを生成する．そして，ミニバッチサイズの長さごとに分割したものを index リストとし，順次 1 つ目のキューに push する．ランダムな整数リストの生成には Chainer が提供する *chainer.iterators.order_samplers.ShuffleOrderSampler* クラスを，ミニバッチサイズ単位の分割には *chainer.iterators._statemachine.iterator_statemachine* メソッドをそれぞれ使用した．

3.2 訓練データプリフェッチ

1 つ目のキューから index リストを pop し，その index リストに含まれる index に対応するファイルを指定されたノードローカルストレージ上のディレクトリにコピーする．この時，ノードローカルストレージ上に同名のファイルが存在しているかどうかをチェックし，存在していたらコピーをスキップする．pop した index リストに含まれる index をすべて処理し終えたら，2 番目のキューに index リストを push する．

3.3 ミニバッチ生成

2 つ目のキューから index リストを pop し，その index リストに含まれる index に対応するファイルをノードローカルストレージ上から read し，ミニバッチを編成する．編成されたミニバッチはメモリ上のキューに格納される．ノードローカルストレージからファイルを read する際は複数のプロセスを使用して並列に read することでスループットを向上させている．ファイルを複数のプロセスで並列に read する処理の実装は Chainer が提供する *chainer.iterators.MultiprocessIterator* の実装から流用した．

4. 評価実験

本章では，提案手法の効果を測定する評価実験の内容と，その結果について説明する．

本論文における評価実験は，すべて筑波大学計算科学センター[*3]にて運用されている HPC クラスタである Cygnus の計算ノードを使用して実施した．Cygnus の計算ノードの仕様を**表 1** に示す．Cygnus の計算ノードにはノードローカルストレージとして 3.2TB の容量を持つ NVMe SSD が搭載され，外部ストレージとして Lustre で構成された分散ファイルシステムが提供されている．評価実験で使用した Python のバージョンは 3.6.8，CUDA のバージョンは 10.1，cuDNN のバージョンは 7.5.0，NCCL のビルドバージョンは 2402，OpenMPI のバージョンは 3.1.4 である．

提案手法の効果を確認するために，以下に述べる 2 種類の評価実験を実施した．比較対象として，Chainer が提供する *chainer.iterators.MultiprocessIterator* クラスを用いて計測した結果をベースラインとして使用する．このクラスは，提案手法における「ミニバッチ生成」ステージに該当する処理のみを行うように実装されている．提案手法としての相違点として，index リストを生成するループ処理が含まれていない点，プリフェッチを行わない点が挙げられる．このクラスを使用し，Cygnus の外部ストレージである Lustre，およびノードローカルストレージである NVMe SSD に訓練データを配置する 2 パターンを使用する．SSD を使用するパターンは理想のケースとして位置づけ，Lustre と SSD を組み合わせて使用する提案手法がどこまで SSD を使用するパターンの結果に近づけるかを評価する．**図 3** にベースラインにおける訓練データの流れを示す．この図の (A) がノードローカルストレージに配置する場合であり，(B) が外部ストレージに配置する場合を示す．

4.1 1,000 ミニバッチバッチ生成ベンチマーク

まず，提案手法における read I/O のバンド幅を計測するため，ミニバッチ生成に要する時間のみを計測した．評価方法は以下の通りである．使用したデータは，Imagenet Large Scale Visual Recognition Challenge 2012 で使用された訓練用の画像データセットから，グレー画像を除外して抽出した 1,261,197 枚である．このデータを使用し，32 ファイルから構成されるミニバッチが 1,000 個生成されるまでの時間を，バッチ生成ステージのプロセス数を変えながら 10 回計測し，その平均値を計測値として使用した．この評価実験はシングルノードで実行し，ミニバッチ生成ステージのプロセス数は 2,4,8,16 を使用した．提案手法における訓練データプリフェッチステージのプロセス数は固定で 2 を使用している．評価結果を**図 4** に示す．

ベースラインにおいては，SSD と Lustre の両ケースで，ミニバッチ生成ステージのプロセス数の増加に伴って処理時間が短縮している傾向が確認された．プロセス数が 12,16 になると，ベースラインにおける SSD を用いた結果

[*1] https://github.com/chainer/chainer
[*2] 本稿執筆時点では Python が 76.9%，C++が 20.2%である．
[*3] https://www.ccs.tsukuba.ac.jp/

表 1　Cygnus 計算ノード仕様

CPU	Intel Xeon Gold 6126 Processor (12C 2.6GHz) × 2 基
Memory	192GiB (16GiB DDR4-2666 ECC RDIMM × 12 基)
GPU	NVIDIA Tesla V100 32GiB HBM2 PCIe 3.0 × 4 基
ネットワーク	InfiniBand HDR100 × 4
NVMe SSD	Intel SSD PC P4610 Series 3.2TB × 1 基
OS	CentOS Linux release 7.6.1810

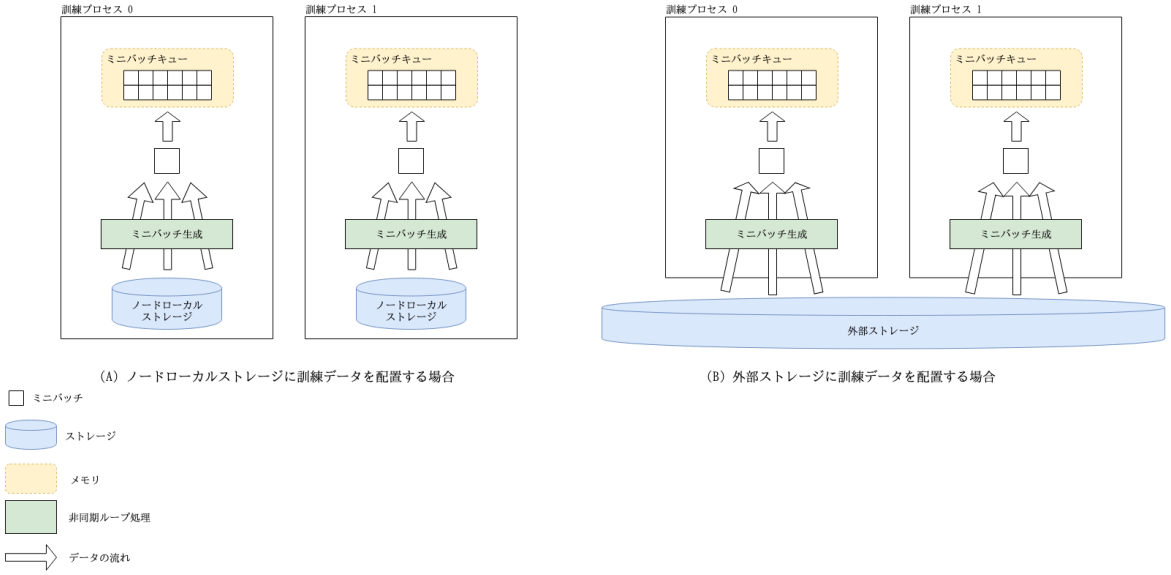

図 3　ベースラインにおける訓練データの流れ

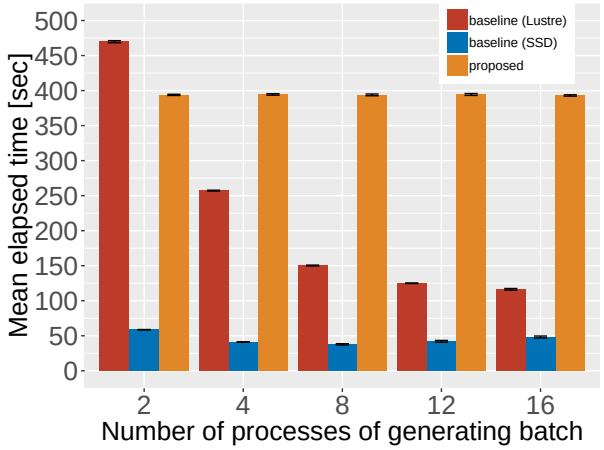

図 4　1,000 ミニバッチ生成ベンチマークの全評価結果．
横軸はミニバッチ生成ステージのプロセス数を示す．

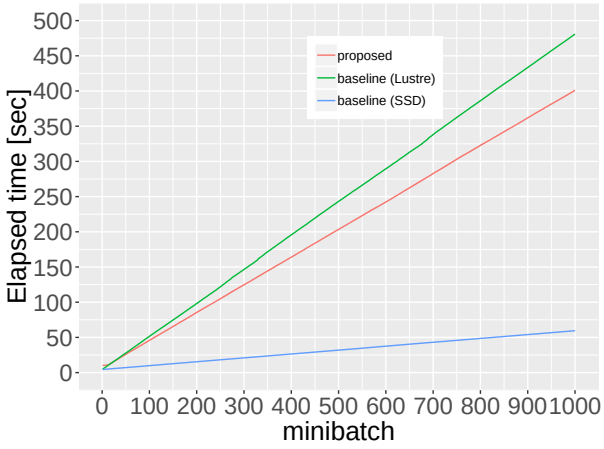

図 5　1,000 ミニバッチ生成ベンチマークの評価結果におけるミニバッチ生成ステージのプロセス数が 2 の時の所要時間の推移．
横軸はベンチマークによってロードしたその時点でのミニバッチの総数を示す．

においては逆に処理時間が増加している現象が確認された．これはプロセスを fork するコストが並列化による処理時間の短縮効果を上回ったためと考えられる．

ミニバッチ生成ステージがのプロセス数が 2 の時の処理時間の推移を，**図 5** に示す．このグラフは横軸がその時点でのロードしたミニバッチの総数，縦軸がその時点での所要時間を示す．また，1 ミニバッチ目のロードに要した所要時間を**表 2** に示す．提案手法とベースラインと Lustre の組み合わせを比較すると，ロードしたミニバッチの総数が 1 から 35 まではベースラインと Lustre の組み合わせの方が処理時間が短いが，それ以降は提案手法が逆転している．これは，提案手法はベースラインよりも生成するプロセス数が多いため，初期化時に fork するコストが大きいため，表 2 に示すように提案手法の 1 ミニバッチ目のオー

表 2　ミニバッチ生成ステージのプロセス数が 2 の時の 1 ミニバッチ目のロードに要した所要時間

	first minibatch [sec] (ratio [%])	全体 [sec]
ベースライン + SSD	4.38 (7.36)	59.49
ベースライン + Lustre	4.77 (0.99)	480.85
提案手法	10.12 (2.5)	400.88

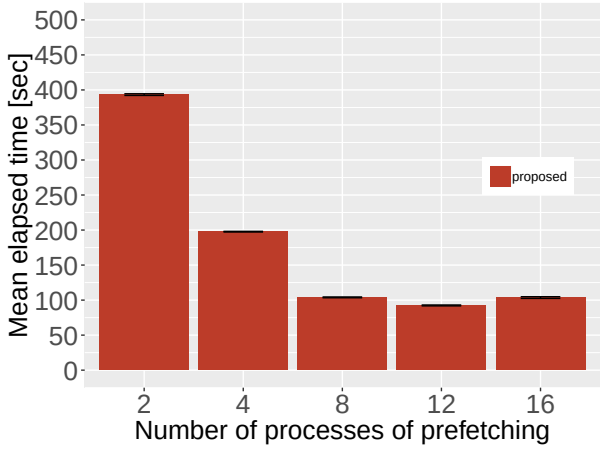

図 6　1,000 ミニバッチ生成ベンチマークにおいて，提案手法のプリフェッチステージのプロセス数を変えて計測した評価結果．横軸はプリフェッチステージのプロセス数を示す．全ての計測結果において，ミニバッチ生成ステージのプロセス数は 2 である．

バーヘッドがベースラインよりも大きくなることによるものである．それ以降は，外部ストレージからのプリフェッチにより，ミニバッチ単位のロード時間はベースラインよりも短くなるため，最終的にベースラインよりも所要時間が短くなっている．

一方で，提案手法についてはミニバッチ生成ステージのプロセス数が変化しても処理時間はほぼ一定で変化していない．これは，プリフェッチステージのプロセス数が 2 で固定のため，プリフェッチステージの処理時間が全体のボトルネックとなっているためと考えられる．この仮説を検証するため，提案手法においてプリフェッチステージのプロセス数を 2,4,8,12,16 に変えながら同じ評価を行った．その評価結果を**図 6** に示す．このグラフが示すように，プリフェッチステージのプロセス数を 2 以上に増加させることで，プロセス数の増加に伴って処理時間の削減が見られた．プロセス数 12 と 16 における結果を比較すると，プロセス数 16 の方が処理時間が長くなっている．これは，ベースラインで SSD を使用したケースと同様にプロセスを fork するコストによるものと考えられる．

結果として，ベースラインと SSD の組み合わせにおいてはミニバッチ生成ステージのプロセス数が 8 のときに最速で 37.74 秒，ベースラインと Lustre の組み合わせにおいてはミニバッチ生成ステージのプロセス数が 16 のときに最速で 116.35 秒，提案手法においてはプリフェッチステージのプロセス数が 12，ミニバッチ生成ステージのプロセス数が 2 のときに最速で 92.4 秒という結果となった．よって，提案手法はベースラインと Lustre との比較において，最大で約 20%高い read I/O を示した．

使用したプロセス数の合計値という観点で見ると，ベースラインと Lustre の組合わせよりも提案手法の方がより少ないプロセス数でより高いバンド幅を実現しており，read I/O のバンド幅の観点においては，ベースラインと Lustre の組合せよりも提案手法が有効な手法であることを示した．

4.2　10 epoch データ並列訓練

次に，実際に深層ニューラルネットワークの訓練処理を 10 epoch 行い，訓練が完了するまでの時間を計測した．評価方法は次の通りである．使用するデータは 1,000 ミニバッチバッチ生成ベンチマークで使用したデータと同じデータであり，1 つのミニバッチサイズに含まれるファイル数も同じく 32 個である．訓練処理に使用するネットワークは畳み込みニューラルネットワークの 1 つである ResNet50[13] を使用し，実装は Chainer に付属するサンプルスクリプト[*4]をベースとして改良したものを使用した．また，集団通信のレイテンシを削減するため，NVIDIA Collective Communications Library (NCCL)[14] を使用するオプションを適用した．ノード数は 1,2,4,8,16 を使用し，それぞれのノード上で動作させる Chainer のプロセス数は固定で 4 である．これは，各計算ノード上には GPU が 4 基搭載されているため，Chainer のプロセス 1 つに対し 1 基の GPU を割り当てるためである．各 Chainer のプロセスにおいて，バッチ生成ステージのプロセス数，訓練データプリフェッチステージのプロセス数は，共に固定で 2 を使用している．これは，各計算ノードに搭載されている CPU のコア数が 12 個であるため，各ノード内で生成されるプロセス数の合計値が，CPU コア数を超えないようにするためである．

評価結果を**図 7** に示す．このグラフはノード数別の 10 epoch 訓練にの所要時間を示したものである．また，各手法における詳細な計測結果については付録 A.1 に示す．このグラフが示すように，全体的にノード数の増加とともに所要時間が減少していることが確認できる．手法別に結果を比較すると，ノード数 1,2,4 においては，ベースラインと SSD，提案手法，ベースラインと Lustre の順に所要時

[*4] https://github.com/chainer/chainer/blob/master/examples/chainermn/imagenet/train_imagenet.py

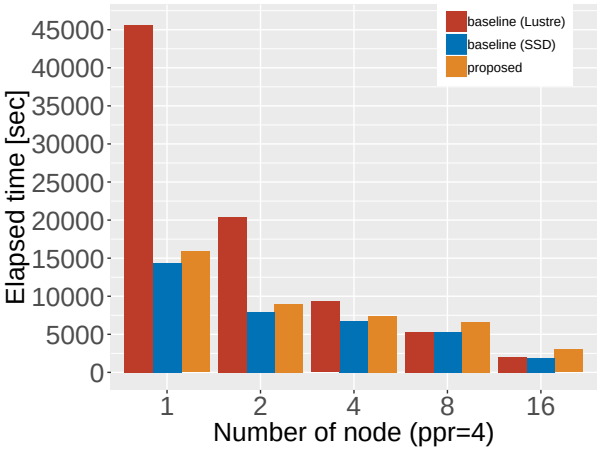

図 7　10 epoch の訓練に要した処理時間

間が短い．これは先の 1,000 ミニバッチ生成ベンチマークの結果と一致する．一方で，1,000 ミニバッチ生成ベンチマークの結果と比べると，ベースラインと SSD の組み合わせと提案手法の処理速度の差が小さくなっている．また，8 ノード目，16 ノード目にベースラインと Lustre の組み合わせによる結果が提案手法，ベースラインと SSD の組み合わせを上回る速度を示している．これら理由については，4.4 節にて詳細に議論する．

一方で，ノード数 8 においては，ベースラインと Lustre の組み合わせの方が提案手法よりも高速であり，ノード数 16 においてはベースラインと SSD の組み合わせよりも高速という結果となっている．他に阻害要因がなければ，1,000 ミニバッチ生成ベンチマークの結果で示したように，read I/O 性能だけが律速要因となるため，実際の訓練処理時においてもこの順序に変化は生じないと考えられる．そのため，実際には別の阻害要因が存在していると考えられる．

4.3　訓練データのファイルコピー時間を加味した評価

先に述べたように，ベースラインと SSD の組み合わせにおいては実際には訓練処理の開始前に訓練データセットのファイルコピーを行っている．そのため，ベースラインと SSD の組み合わせの評価結果にファイルコピーに要した時間を加味した上での比較を行う．

本評価実験のベースラインと SSD の組み合わせにおいては，訓練処理開始前に，各ノードで tar 形式でアーカイブした訓練データを外部ストレージからノードローカルストレージにコピーして解凍している．なお，訓練データセットを外部ストレージ上で tar 形式にアーカイブしたところ，約 6 時間を要した．一方，提案手法においては，訓練処理中に 1 ノードあたり合計 8 プロセスで外部ストレージからノードローカルストレージへファイルコピーを行っている．そのため，条件を合わせるためにはアーカイブされていない訓練データセット全てを 8 並列でコピーする時間と

表 3　各ノードにおいて 8 並列でファイルをコピーした際の所要時間

ノード数	所要時間 [sec]
1	1,749.275
2	2,031.625
4	1,575.716
8	1,592.248
16	1,612.894

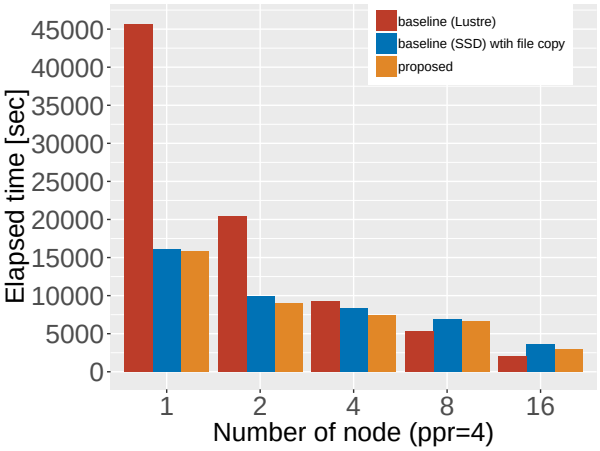

図 8　10 epoch の訓練に要した処理時間において，ベースライン（SSD）にファイルコピーに要する時間を追加した結果

比較することが妥当であると考える．そこで，今回使用した訓練データセット 1,261,197 枚の画像を，GNU parallel コマンドを用いてディレクトリ単位で 8 並列でコピーし，全ノードでコピーが完了するまでの時間を time コマンドで計測した．計測結果を**表 3** に示す．ノード数の増加に伴って Lustre への read I/O は増加するが，今回の計測ではノード数と所要時間の間には相関性は見られなかった．これは，実行された計算ノード上の Page cache の状況や Lustre 側のキャッシュの状況に左右されていると考えられる．これらの所要時間を図 7 におけるベースラインと SSD の組み合わせの結果に追加した結果を**図 8** に示す．

このグラフが示すように，全ケースにおいてファイルコピーに要すると想定される時間を加味することで，ベースラインと SSD の組合わせよりも所要時間が短くなるという結果を示している．

4.4　データ並列訓練における処理時間の内訳

この節では，epoch 毎に訓練に要した処理時間の内訳を調査し，手法ごとの傾向の差と，その理由に関して議論する．

図 9，**図 10**，**図 11** に，提案手法における 10 epoch 訓練時の処理時間の内訳をそれぞれ示す．

これは 16 ノード/64 プロセスで 10 epoch 訓練を行った際の各処理内容における処理時間を epoch 別に合計したものである．各棒グラフの色分けは，上から AllReduce，backward，forward，ミニバッチのロード，その他の所要

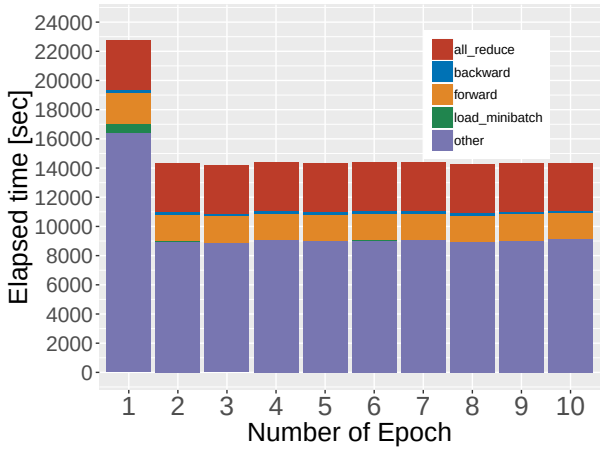

図 9　提案手法における 10 epoch 訓練時の各 epoch における所要時間の内訳．ノード数は 16，プロセス数は 64 である．

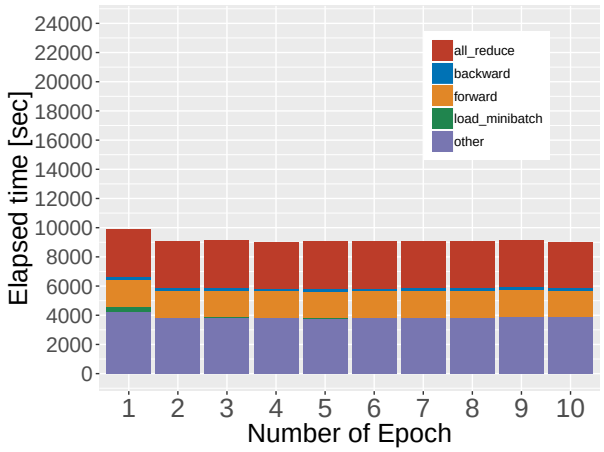

図 10　ベースラインと SSD の組み合わせおける 10 epoch 訓練時の各 epoch における所要時間の内訳．ノード数は 16，プロセス数は 64 である．

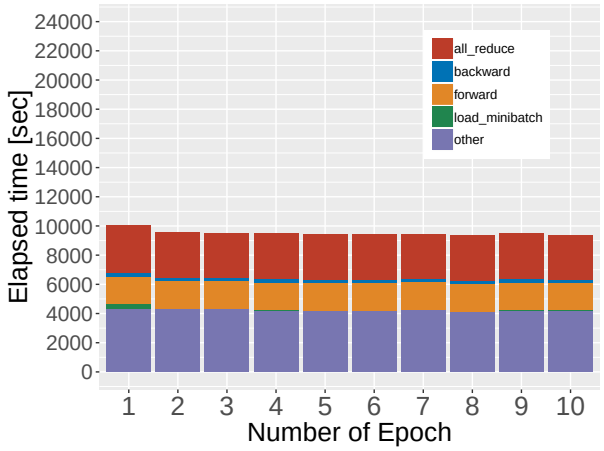

図 11　ベースラインと Lustre の組み合わせおける 10 epoch 訓練時の各 epoch における所要時間の内訳．ノード数は 16，プロセス数は 64 である．

時間をそれぞれ示す．

全ての集計結果に共通する傾向として，1 epoch 目はミニバッチのロードに一定の所要時間を要しているが，2 epoch 目からはほとんど所要時間が発生していない．これは，1 epoch 目の処理中にミニバッチをバッファリングするスピードが 1 iteration の訓練速度を上回り，2 epoch 目以降はほとんどの iteration でメモリ上にバッファされたミニバッチをロードするようになった結果と考えられる．

1 epoch 目におけるミニバッチのロード時間の分布を調査するため，1 epoch 目の各 iteration におけるミニバッチのロード時間を調査した．その結果，1 iteration 目にほとんどの所要時間が集中していることが判明した．1 epoch 目の 1, 2iteration におけるミニバッチのロードに要した時間を**表 4** に示す．この表に示すとおり，1 epoch 目におけるミニバッチのロードに要した時間は，どの手法においても 90%以上が 1 iteration 目のロード時間によって占められている．これは，1 iteration 目ではベースラインと提案手法の両方においてプロセスの fork などの初期化処理が含まれ，またメモリ上のバッファにミニバッチが 1 つもバッファされていない．そのため，他の iteration に比べて極端に所要時間が長くなっている．この現象を時系列で表した図を**図 12** に示す．この図は 1 iteration 目からの処理状況を時系列で示したものである．上部の矢印は訓練プロセスの各フェーズの処理時間を示したものであり，下部の矢印は訓練データからミニバッチを生成するプロセスの処理時間を示したものである．1 iteration 目においては訓練プロセス側でミニバッチのロードに対する待ち時間が発生するが，2 iteration 目においてはすでにミニバッチバッファ上にミニバッチが存在しているので必要な所要時間はメモリへのアクセス時間のみとなっている．そのため待ち時間が発生しない．それ以降の iteration においても同じ状況が続くため，結果的に 1 iteration 目にほとんどのミニバッチのロードに要する処理時間が集中する結果となったと考えられる．

1 epoch 目の所要時間を手法間で比較すると，提案手法が 683.31 秒と最も長く時間がかかっている．これは，1,000 ミニバッチ生成ベンチマークにおいても示したように，提案手法は生成するプロセス数がベースラインよりも多いため，その分オーバーヘッドが大きいためである．ベースラインにおいては，SSD よりも Lustre の方が所要時間が短くなっているが，これは繰り返し評価を行ったことによる計算ノード上の Page cache による影響と考えられる．実際に，複数回同様の評価実験を行うと，ベースラインにおいては Lustre と SSD の結果が逆転するケースも確認された．

また，2 iteration 目は 1 iteration 目と比べると大幅に所要時間が削減している．これは，1 iteration 目のミニバッチのロードが完了した後，訓練処理を行っている間に非同期でミニバッチをメモリ上にバッファしているため，2 iteration 目以降のミニバッチのロード時には，すでにミニバッチがバッファ上に存在している状態となっているためと考えられる．そのため，2 iteration 目においては提案手

表 4　1 epoch 目の 1, 2iteration 目におけるミニバッチのロードに要した全プロセスの合計時間

	first iteration [sec] (ratio [%])	second iteration [sec] (ratio [%])	1 epoch 全体 [sec]
ベースライン + SSD	**334.55 (92.9)**	0.2450 (0.7)	360.047
ベースライン + Lustre	**289.92 (98.5)**	0.0048 (0.002)	293.78
提案手法	**683.31 (99.8)**	0.0027 (0.0004)	684.94

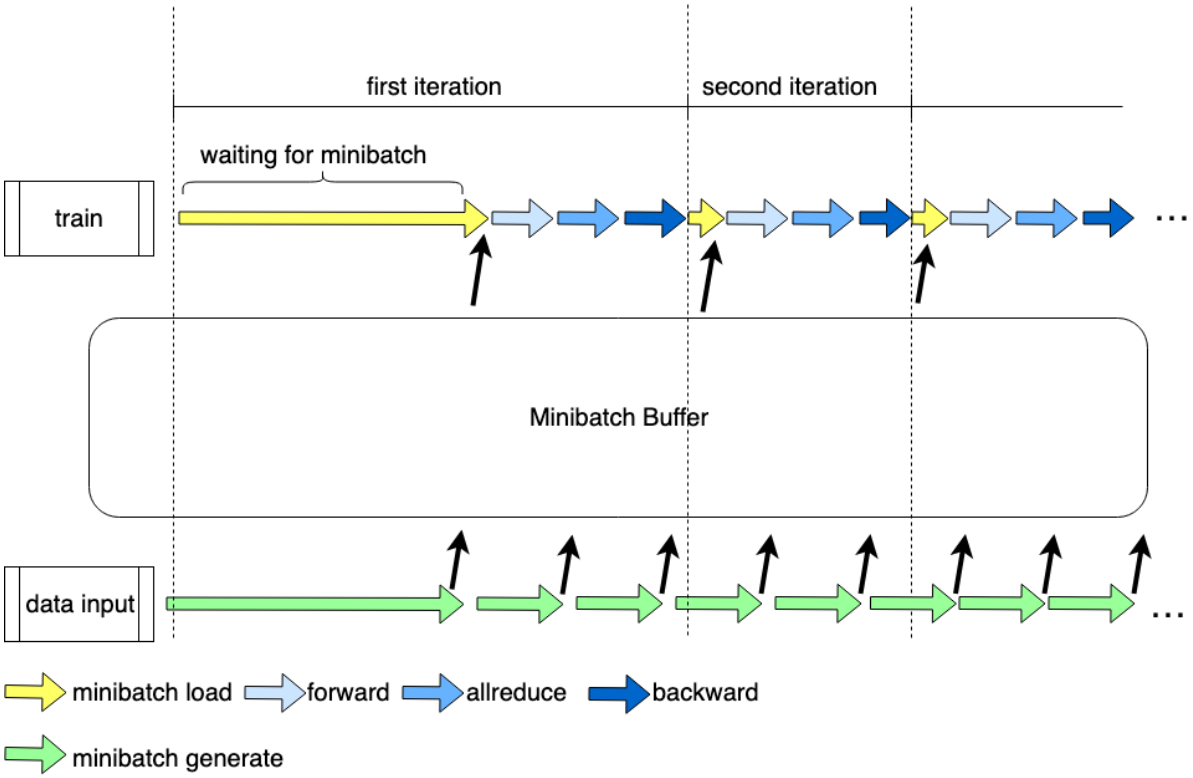

図 12　データ並列訓練時における訓練プロセスと提案手法/ベースラインによる訓練データ入力処理の状況を時系列で表した図.
1 iteration 目はオーバーヘッドがあるため訓練プロセス側に待ち時間が発生するが, 2 iteration 目以降はオーバーヘッドがなくなりミニバッチ生成までの処理時間が訓練プロセスの処理時間よりも短くなるため，ミニバッチがすでにバッファ上に存在する状態になるので待ちが発生しなくなる.

法とベースラインとの間における結果の差がほとんどなくなっていると考えられる．1,000 ミニバッチ生成ベンチマークの結果が示す通り，もし read I/O が律速する状況が継続していれば，提案手法の方がベースラインと Lustre との組み合わせよりもミニバッチロードの所要時間が短い結果になっていたと考えられる．しかし，実際の訓練処理においては，read I/O ではなく GPU 側の処理と AllReduce 等の処理が全体の所要時間を律速している状況となっていた．そのため，今回の評価実験においては，手法間の差が見られなくなっていると考えられる．

一方で，各 epoch の全体の所要時間に注目すると，提案手法とベースラインにおいて，1 epoch 内の所要時間に占める「その他」の部分の割合が大きく異なっている．ベースラインにおいては，Lustre では 1 epoch 目では約 43%，2 epoch 目では約 45%であり，SSD では 1 epoch 目では約 43%，2 epoch 目では約 42%である．一方，提案手法においては，1 epoch 目では 71%，2 epoch 目では約 63%を占めており，ベースラインよりもかなり大きくなっている．「その他」の部分で行われているのは CPU 上で行われる処理であり，load したミニバッチの前処理や，各種メソッドの呼び出しに要する所要時間で構成されている．

提案手法とベースラインとの間で「その他」の所要時間が占める割合の差が生じている原因として，訓練処理中に動作しているプロセス数の差が影響していると考えられる．提案手法においては，index リスト生成ステージに 1 プロセス，訓練データプリフェッチステージに 2 プロセス，ミニバッチ生成に 2 プロセスを割り当て，それに加えて Chainer のメインプロセスが 1 プロセス動作するため，合計で 6 プロセスが動作している状態となる．本論文では，常に 1 ノード当たりの Chainer プロセス数は 4 プロセスで評価している

ため，1 ノード当たりで訓練処理で使用する合計プロセス数は，$6 \times 4 = 24$ プロセスとなる．評価環境の Cygnus の計算ノードに搭載された CPU に搭載された CPU コア数は 12 個であるため，搭載された CPU を 2 ソケットとも使い切っている状態であると考えられる．Cygnus の計算ノード上で，Python の *multiprocessing* module において利用可能な CPU コア数を表示する *multiprocessing.cpu_count*() メソッドを用いて利用可能な CPU コア数を表示すると「24」と表示されるため，Python からは 2 ソケットに搭載された CPU コアすべてが利用可能な状態になっていると考えられる．そのため，提案手法において使用された 24 プロセスは，それぞれ個別の CPU コアに 1:1 で割り当てられて動作する状態になっていたと考えられる．一方，ベースラインでは，ミニバッチ生成ステージに 2 プロセスを割り当てているのみなので Chainer プロセスは合計 3 プロセスであるため，1 ノード当たり $3 \times 4 = 12$ プロセスである．これは 1 ソケットに搭載された CPU コア数と等しい．本論文では，評価実験の際にソケット単位での CPU 利用率を計測していないため，各 CPU コアにどのようにプロセスが割り当てられたかは明らかではない．しかし，提案手法とベースラインの間で，プロセスの割当方法が異なることが想定されるため，この差分が CPU 上での処理時間に影響を与えた可能性が考えられる．

また，図 9 に示すように，提案手法において 1 epoch 目の方が「その他」の割合が他の epoch と比べて高くなっている．これは，fork するプロセス数がベースラインよりも多いため，それに伴って 1 epoch 目に実行されるプロセス fork の負荷がベースラインよりも高くなっているためと考えられる．

5. 関連研究

Zhu ら [15] は，データ並列訓練において read I/O を高速化するための分散キャッシュフレームワークである「DeepIO」を提案した．この手法では，訓練処理を行う複数の計算ノード上で，分割された訓練データを非同期に read してメモリ上にキャッシュし，さらにメモリ上のキャッシュからミニバッチを編成し，ミニバッチ単位で別途バッファリングを行うことで read I/O を高速化している．また，RDMA を用いて 1 epoch ごとに訓練データをノード間で交換することで，ミニバッチに含まれるデータの多様性を維持し，汎化性能の劣化を防止している．本論文との共通点として，訓練データを外部ストレージからキャッシュしている点，ミニバッチ単位でバッファリングする点，非同期にデータをキャッシュする点が挙げられる．本論文との相違点として，本論文ではメモリへのキャッシュではなく，ノードローカルストレージへのキャッシュを選択した．この選択の理由は，一般にメモリサイズよりもノードローカルストレージのサイズの方が十分に余裕があるという点によるものである．本論文で評価実験に使用した Cygnus においては，各計算ノード上で約 2.8TiB のノードローカルストレージが提供されているが，メモリは 192GiB であり約 14 倍の差がある．

Chien ら [16] は，TensorFlow における read I/O の高速化機構である Input pipeline の性能評価を行い，訓練データを read する際のプリフェッチによる read I/O 性能への影響を評価している．独自のベンチマークを用いた評価では，Input pipeline において訓練データのプリフェッチを行う場合と行わない場合とで最大 2.3 倍 read I/O を高速化できることを示した．本論文では Chainer を用いて実装しているが，異なるフレームワークを用いた場合であっても訓練データのプリフェッチが訓練データの read I/O 性能に大きく影響することを示している．

6. おわりに

本論文では，外部ストレージとノードローカルストレージの両方を用いて深層ニューラルネットワークの訓練処理における read I/O を高速化する手法を提案し，提案手法を機械学習フレームワークである Chainer に実装した．評価実験として，Chainer が提供する *chainer.iterators.MultiprocessIterator* をベースラインとして採用し，訓練データセットを Lustre，SSD それぞれに配置したケースを対象に read I/O 性能の比較を行った．結果として，1,000 個のミニバッチを生成するのに要する時間を比較したベンチマークでは，提案手法はベースラインと Lustre の組み合わせとの比較において，最大で約 20%高い read I/O 性能を示した．10 epoch のデータ並列訓練における処理時間の比較では，訓練データセットをノードローカルストレージにコピーする時間を加味した場合は，ベースラインと SSD の組合せよりも提案手法の方が訓練処理時間が高速になるケースがあることを示し，しかし，本論文の評価ではデータ並列訓練においては訓練データの read 以外の AllReduce などの時間が訓練処理を律速しているため，ストレージ間の read I/O の性能差は明確には確認できなかった．read I/O の性能差をより正確に評価するためには，データ並列訓練においては，read I/O 以外の処理時間を最適化した上で評価を行う必要性が確認された．

謝辞 本研究の一部は，JST CREST JPMJCR1414，JSPS 科研費 17H01748，国立研究開発法人新エネルギー・産業技術総合開発機構（NEDO）および富士通研究所との共同研究の助成を受けたものである．

参考文献

[1] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C. and Fei-Fei, L.: ImageNet

Large Scale Visual Recognition Challenge, *International Journal of Computer Vision (IJCV)*, Vol. 115, No. 3, pp. 211–252 (online), DOI: 10.1007/s11263-015-0816-y (2015).

[2] Abu-El-Haija, S., Kothari, N., Lee, J., Natsev, P., Toderici, G., Varadarajan, B. and Vijayanarasimhan, S.: YouTube-8M: A Large-Scale Video Classification Benchmark, *CoRR*, Vol. abs/1609.08675 (online), available from ⟨http://arxiv.org/abs/1609.08675⟩ (2016).

[3] Kurth, T., Treichler, S., Romero, J., Mudigonda, M., Luehr, N., Phillips, E., Mahesh, A., Matheson, M., Deslippe, J., Fatica, M., Prabhat and Houston, M.: Exascale Deep Learning for Climate Analytics, *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, Piscataway, NJ, USA, IEEE Press, pp. 51:1–51:12 (online), available from ⟨http://dl.acm.org/citation.cfm?id=3291656.3291724⟩ (2018).

[4] Simonyan, K. and Zisserman, A.: Very Deep Convolutional Networks for Large-Scale Image Recognition, *CoRR*, Vol. abs/1409.1556 (online), available from ⟨http://arxiv.org/abs/1409.1556⟩ (2014).

[5] 芹沢 和洋，建部修見：深層ニューラルネットワークにおける訓練高速化のための自動最適化，情報処理学会第 168 回 HPC 研究会報告 (HPC168)，Vol. 2019-HPC-168, No. 25, （オンライン），入手先 ⟨http://id.nii.ac.jp/1001/00194707/⟩ (2019).

[6] Tokui, S., Oono, K., Hido, S. and Clayton, J.: Chainer: a Next-Generation Open Source Framework for Deep Learning, *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, (online), available from ⟨`http://learningsys.org/papers/LearningSys_2015_paper_33.pdf`⟩ (2015).

[7] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M. et al.: Tensorflow: Large-scale machine learning on heterogeneous distributed systems, *arXiv preprint arXiv:1603.04467* (2016).

[8] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., aurelio Ranzato, M., Senior, A., Tucker, P., Yang, K., Le, Q. V. and Ng, A. Y.: Large Scale Distributed Deep Networks, *Advances in Neural Information Processing Systems 25* (Pereira, F., Burges, C. J. C., Bottou, L. and Weinberger, K. Q., eds.), Curran Associates, Inc., pp. 1223–1231 (online), available from ⟨http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf⟩ (2012).

[9] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K. and Fei-Fei, L.: ImageNet: A Large-Scale Hierarchical Image Database, *CVPR09* (2009).

[10] Yamazaki, M., Kasagi, A., Tabuchi, A., Honda, T., Miwa, M., Fukumoto, N., Tabaru, T., Ike, A. and Nakashima, K.: Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds, *CoRR*, Vol. abs/1903.12650 (online), available from ⟨http://arxiv.org/abs/1903.12650⟩ (2019).

[11] Mikami, H., Suganuma, H., U.-Chupala, P., Tanaka, Y. and Kageyama, Y.: ImageNet/ResNet-50 Training in 224 Seconds, *CoRR*, Vol. abs/1811.05233 (online), available from ⟨http://arxiv.org/abs/1811.05233⟩ (2018).

[12] Ying, C., Kumar, S., Chen, D., Wang, T. and Cheng, Y.: Image Classification at Supercomputer Scale, *CoRR*, Vol. abs/1811.06992 (online), available from ⟨http://arxiv.org/abs/1811.06992⟩ (2018).

[13] 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 (2016).

[14] : NVIDIA Collective Communications Library (NCCL), `https://developer.nvidia.com/nccl`.

[15] Zhu, Y., Chowdhury, F., Fu, H., Moody, A., Mohror, K., Sato, K. and Yu, W.: Entropy-Aware I/O Pipelining for Large-Scale Deep Learning on HPC Systems, pp. 145–156 (online), DOI: 10.1109/MASCOTS.2018.00023 (2018).

[16] W. D. Chien, S., Markidis, S., Prasad Sishtla, C., Santos, L., Herman, P., Narasimhamurthy, S. and Laure, E.: Characterizing Deep-Learning I/O Workloads in TensorFlow (2018).

付　　録

A.1　10 epoch 訓練における評価結果詳細

表 A·1 ベースライン（SSD）において 10 epoch 訓練に要した時間（ppr=4 で固定）
ミニバッチ 1 個のデータサイズは平均 1.31MiB と仮定している．
各ノードのバンド幅の計算方法は，1 ミニバッチのデータサイズ × 合計 iteration 数 × ppr である．

ノード数	合計訓練時間 [sec]	合計 iteration 数	iteration/sec	各ノードの推定バンド幅 [MiB/s]
1	14,339.98	98,532	6.87	36.00
2	7,924.18	49,266	6.22	32.58
4	6,716.36	24,633	3.67	19.22
8	5,294.47	12,317	2.33	12.19
16	1,938.19	6,159	3.18	6.29

表 A·2 ベースライン（Lustre）を用いて 10 epoch 訓練に要した時間（ppr=4 で固定）
ミニバッチ 1 個のデータサイズは平均 1.31MiB と仮定している．
全ノードのバンド幅の計算方法は，1 ミニバッチのデータサイズ × 合計 iteration 数 × Chainer プロセス数である．

ノード数	合計訓練時間 [sec]	合計 iteration 数	iteration/sec	全ノードの推定バンド幅 [MiB/s]
1	45,586.36	98,532	2.16	11.33
2	20,388.48	49,266	2.42	25.32
4	9,293.58	24,633	2.65	55.56
8	5,290.22	12,317	2.33	97.6
16	2,055.71	6,159	3.00	116.78

表 A·3 提案手法を用いて 10 epoch 訓練に要した時間（ppr=4 で固定）
ミニバッチ 1 個のデータサイズは平均 1.31MiB と仮定している．
各ノードのバンド幅の計算方法は，1 ミニバッチのデータサイズ × 合計 iteration 数 × ppr である．

ノード数	合計訓練時間 [sec]	合計 iteration 数	iteration/sec	各ノードの推定バンド幅 [MiB/s]
1	15,878.37	98,532	6.21	32.52
2	9,009.86	49,266	5.47	28.65
4	7,363.48	24,633	3.35	17.53
8	6,585.40	12,317	1.87	9.80
16	3,008.96	6,159	2.05	4.71