Vol.2021-HPC-182 No.7
2021/12/6

深層学習における実行時ファイルステージング

樋口 遼太郎[1,a)]　三輪 忍[1]　八巻 隼人[1]　本多 弘樹[1]

概要：外部ストレージを利用してファイル共有を行う高性能計算機システムにおいては、複数の計算ノードがファイルアクセスを行う際に外部ストレージに対する I/O 競合が発生し、ファイルアクセス性能が低下することがある。この問題を緩和する方法としてファイルステージングがあるが、従来のステージング手法は、深層学習アプリケーションのような全実行時間に占めるファイルアクセス時間の割合が大きいアプリケーションに対してあまり効果的でない。そこで本稿では、アプリケーション実行と並列にファイルステージングを行う手法を提案する。提案手法では深層学習アプリケーション内でステージング処理を呼び出し、学習処理と並列化することでステージング処理に要する時間を隠蔽する。TSUBAME3.0 と 3 種類の TensorFlow アプリケーションを用いて評価した結果、提案手法は従来手法に対して、I/O 競合時の実行時間を最大 30.8%削減できていることが確認できた。

キーワード：ファイルステージング、深層学習、TensorFlow

1. はじめに

高性能計算機システムの多くは計算ノード外部に大容量のストレージを有しており、この外部ストレージを用いて計算ノード間でファイルを共有している。外部ストレージと計算ノード間は一般には InfiniBand 等の高速・大容量な専用ネットワークで接続されており、これにより高いファイルアクセス性能を実現する。しかしながら、複数の計算ノードが集中的にファイルアクセスを行う状況（例えば複数のユーザが I/O インテンシブなジョブを実行する状況）においては、外部ストレージに対するアクセス競合が発生し、ファイルアクセス性能が著しく低下する。ファイル I/O 競合が発生している状況で新たなジョブを実行すると、上記のファイルアクセス性能低下が原因でジョブの実行時間は通常よりも大幅に増加してしまう。

上記の問題を緩和する方法の 1 つにファイルステージングがある。ファイルステージングは、必要なファイルを外部ストレージから計算ノード内のローカルストレージ（SSD 等によって構成された高速ストレージ）にコピーした上で、ローカルストレージ上のファイルを用いてアプリケーションの実行を行う技術である。外部ストレージからローカルストレージへのファイルコピー処理は、通常はジョブスクリプト内に記述され、アプリケーションの実行開始に先立って行われる。ステージングを行った場合はアプリケーションの実行中に外部ストレージに対するアクセスが発生しないため、外部ストレージにおけるファイル I/O 競合の影響を受けることなくアプリケーションを実行できる。

表 1: 外部ストレージの負荷と深層学習の実行時間 [s]

	低負荷時	高負荷時
ステージングなし	3478	4252
従来のステージング	3693	5189

一方、従来のファイルステージングはファイルコピーとアプリケーション実行を逐次的に行うため、全実行時間に占めるファイルアクセス時間の割合が大きいアプリケーションに対してあまり効果がないという問題がある。そのようなアプリケーションの 1 つに深層学習アプリケーションがある。例えば文献 [8], [9] では、大規模な訓練データに対して深層学習を行うケースにおいて、訓練データファイルのリード性能がアプリケーション全体の実行時間を左右することが報告されている。加えて、本稿の事前評価として、ファイル I/O が低負荷と高負荷の時に深層学習アプリケーションを実行した場合の計測結果を**表 1** に示す。表 1 が示す通り、ファイルステージングを用いずに実行した場合と、従来のファイルステージングを用いて実行した場合では、ファイル I/O が高負荷時に実行時間が大きく増加している。

そこで本研究では、外部ストレージからローカルスト

[1] 電気通信大学
1-5-1, Chofugaoka, Chofu, Tokyo 182–8585, Japan
[a)] higuchi@hpc.is.uec.ac.jp

レージへのファイルコピーをアプリケーション実行時に行う手法を提案する。提案手法では、深層学習アプリケーションの実行開始直後に将来アクセスされる予定のファイルのリスト、および、このリストを用いてファイルコピーをバックグラウンドで行うプロセスを生成し、学習処理を行うメインのプロセスと並列にファイルコピーを行う。そして、深層学習アプリケーションはデータセットの学習が完了するたびにファイルのコピー状況を確認し、新たにコピーの完了したファイルがあった場合はそれらのファイルを用いてデータセットを作成し直して学習を行う、という処理を繰り返す。このようにファイルコピーと学習を並列化することで前者の処理に要する時間を隠蔽し、ファイル I/O 競合時における深層学習アプリケーションの実行時間の増加を抑制する。

ローカルストレージへのファイルコピーと学習を並列に行うアイデアは先行研究 [9] によって提案されているが、提案手法は以下の 2 点において先行研究とは異なる。1) 提案手法はファイルステージングに要する時間の隠蔽を目的としており、先行研究とは違って元のアプリケーションの実行結果と異なる結果を出力することはない。2) 提案手法では高速化のためにコピー状況の確認処理を極力減らすなどの実装上の工夫を行っている。

本稿は以下の構成となっている。まず次章では、研究背景として深層学習とファイルステージングについて説明する。続く 3 章では提案手法の概要を説明し、4 章ではその実装を述べる。5 章で評価方法について述べた後、6 章で評価結果を示し、最後 7 章にて本稿をまとめる。

2. 研究背景

2.1 深層学習

深層学習は、近年大きな注目を集めている技術であり、自動運転や医療診断を始めとする幅広い分野で活用されている。ニューラルネットワークは、4 層を超える多層のネットワーク (DNN:Deep Neural Network) を用いることで、その判断・認識精度が飛躍的に向上することが知られている。深層学習はそのようなネットワークを学習する技術であり、認識精度や学習効率を高めるためにさまざまなネットワークモデルが提案されている [7], [10]。

一般に、深層学習には大量の訓練データが使用される。例えば、画像認識精度を競うコンテストとして有名な ILSVRC では、ImageNet と呼ばれる 100GB を越えるデータセットを用いて学習を行う [2]。また、動画像認識を行うコンテストである YouTube-8M Kaggle Challenge では、1.7TB ものデータセットが使用されている [1]。大量のデータを用いて学習を行うことで、ネットワークの認識精度は飛躍的に向上することが知られている。

ネットワークモデルと訓練データの両方が大規模化した結果、深層学習の計算時間は膨大なものとなっており、時には数日から数週間といった時間を要することもある [3]。そのため、深層学習は高性能計算機システム上で複数 GPU を用いて並列に行われることが多い。深層学習の並列化手法はいくつか存在するが、本稿では多くの深層学習フレームワークがサポートしており、またエンドユーザにも広く利用されているデータ並列学習を対象とする。

高性能計算機システム上でデータ並列学習を行う場合は、一般に、各計算ノードは共有ファイルシステム上に存在する訓練データを参照しながら学習を進める。そのため、競合によって共有ファイルシステムのアクセス性能が低下すると、各ノードの処理時間は増加してしまう。

2.2 ファイルステージング

競合によるファイルアクセス性能低下の問題を緩和する方法の 1 つにファイルステージングがある。ファイルステージングは、アプリケーションが使用する外部ストレージ上のファイルを、各計算ノード上のローカルストレージ等の専用のディスク領域にコピー（ステージイン）してから使用する方法である。また、アプリケーションの実行結果を上記のディスク領域へ出力し、実行終了時にディスク領域上のファイルを外部ストレージへコピー（ステージアウト）することで結果を外部ストレージへ反映させる。SSD などの外部ストレージよりも高速なメモリによって専用のディスク領域は構成されることが多いため、ステージイン/ステージアウト処理が別途必要なものの、ステージングを行うことでアプリケーションのファイルアクセス時間は短縮される。また、ステージインを行ったアプリケーションは外部ストレージにアクセスしなくなるため、I/O 競合によるファイルアクセス性能低下の影響を受けずにアプリケーションを実行できるようになる。

ファイルステージングにおけるステージイン/ステージアウトの処理は、通常はユーザによって明示的に指示する必要がある。これを実現する最も単純な方法はジョブスクリプト内に上記の処理を記述することであり、多くの高性能計算サービスにおいて推奨されている。例えば、Linux の copy コマンドによって外部ストレージからローカルストレージにファイルをコピーする処理を追加し、ローカルストレージ上のファイルを使用してアプリケーションを実行するようにジョブスクリプトを書き換えることでステージインを行う。この場合、外部ストレージからローカルストレージへのファイルコピーとアプリケーション実行は逐次的に行われることになる。

2.3 Catwalk

ファイルステージングに関する研究として、Linux 向けのオンデマンドファイルステージングシステムである、Catwalk が提案されている [5]。このシステムでは、glibc 内のファイルアクセスに関する関数をフックすることで、

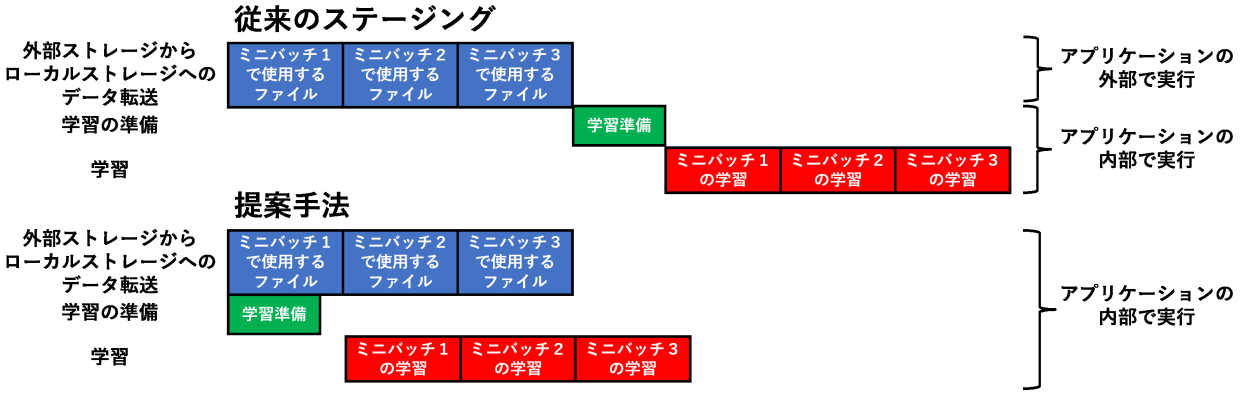


図 1: 従来のステージングと提案手法の違い

リスト 1 従来のステージングにおけるジョブスクリプト内の記述

```
1: copy $External_Dir/data $Local_Dir
2: python Deep_Learning $Local_Dir/data
```

リスト 2 提案手法におけるジョブスクリプト内の記述

```
1: python Deep_Learning $Local_Dir $External_Dir/data
```

アプリケーション内でファイルアクセスが発生した時に当該ファイルがステージインされているか否かを確認し、ステージインされていなかった場合はステージインした後にファイルを開くようにしている。これにより、ユーザによるステージインに関する記述が不要となる。また、ステージイン時のファイルコピー処理をパイプライン化することにより、コピー処理がアプリケーション全体の実行時間に及ぼす影響を軽減している。

Catwalk はアプリケーションの実行中にステージインを行う技術であるが、本研究とは異なり、ファイルコピー時間の隠蔽を目的としたものではない。Catwalk では、アプリケーション内でファイルアクセスが発生した時に当該ファイルのコピー処理を開始するため、当該ファイルのコピーが完了するまではアプリケーション実行が停止することになる。ファイルのコピー時間をアプリケーションの実行時間によって隠蔽するためには、アプリケーションの実行開始時に、実行に使用するすべてのファイルのコピーを開始する必要がある。

3. 実行時ファイルステージング

深層学習アプリケーションの場合、学習に使用する訓練データファイルはアプリケーションの実行開始前に決まっていることが多い。また学習中は、アプリケーションの実行開始直後に決めた順序にしたがって訓練データファイルへのアクセスが繰り返し行われる。そのため、一般的なアプリケーションとは異なり、深層学習アプリケーションにおいてはアプリケーションの実行開始直後に将来アクセスされるファイルの一覧とそのアクセス順序を取得できる。

しかし 1 章で述べたように、従来のファイルステージングを使用し、ファイルコピーとアプリケーション実行を逐次的に行っても、深層学習のような全実行時間に占めるファイルアクセス時間の割合が大きいアプリケーションに対しては、I/O 競合時にあまり効果がないという問題がある。

そこで本稿では、深層学習の実行とステージインを並列に行う実行時ファイルステージングを提案する。2.2 節で述べた従来のステージング手法と提案手法の違いを**図 1** に、従来のステージング手法と提案手法におけるジョブスクリプト内の記述の違いをリスト 1 及びリスト 2 に示す。

従来のステージングでは、外部ストレージからローカルストレージへのファイルコピー処理はジョブスクリプト内で記述され、学習を行うアプリケーション本体とは逐次的に実行される（図 1 上、リスト 1）。それに対して提案手法では、アプリケーションプログラムの修正により深層学習アプリケーションの実行開始後にファイルコピー処理を行うプロセスを別途生成し、アプリケーション本体の実行と並行して所定の順序にしたがってファイルコピーを行う（図 1 下、リスト 2）。また、アプリケーション本体はステージインされたファイルを用いて学習を行い、学習に必要なファイルがステージインされていなければ同ファイルのコピーの完了を待つように修正する。このように提案手法ではファイルコピーと学習をパイプライン並列で処理することにより、I/O 競合時のファイルコピー時間を隠蔽する。

上述のように提案手法では学習に必要なファイルのコピーが完了しているか否かの確認をアプリケーション本体で行う必要があり、この処理が性能オーバーヘッドとなる。そこで提案手法では、このオーバーヘッドを削減するために、ファイルコピーの完了を確認する回数を極力減らす実装を行っている。同実装の詳細は次章で述べる。

従来のステージング手法とは異なり、提案手法はアプリケーションプログラムの修正が必要である。ただし、提案手法では以上で述べた処理を深層学習フレームワークのメソッドを拡張して実装することにより、エンドユーザによるプログラム修正の範囲を最小限にする工夫を行っている。

4. 実装

本稿では、代表的な深層学習フレームワークである TensorFlow に対して提案手法の実装を行った。本章ではその詳細を述べる。

4.1 全体像

従来の深層学習プログラムの全体像をリスト 3 に、提案手法を使用する深層学習プログラムの全体像をリスト 4 に示す。前章で述べたように、提案手法の大部分は TensorFlow のメソッドを拡張する形で実装されている。そのため、エンドユーザは基本的にはプログラム内で呼び出すメソッドの一部を変更あるいは追記するだけで、提案手法を利用できる。

エンドユーザによる追記と変更が必要なメソッドは、具体的にはリスト 4 の Copy（3 行目）, Make_Dataset（7 行目）, Training（8 行目）の 3 種類である。それぞれのメソッドは、外部ストレージからローカルストレージへのファイルコピーを行うプロセスの生成、ローカルストレージへのコピーが完了したファイルから学習に使用するデータセットの作成とコピー完了の待ち合わせ、作成したデータセットを用いた学習の実行を行う。

Copy はプログラムの初期化部分において一度だけ（具体的には、プログラムの入力引数として与えられた訓練データファイルのリストをシャッフルし、学習時のアクセス順序が定まった直後に）呼び出される。一方、Make_Dataset と Training は通常は一度だけ呼び出される処理であるが、提案手法ではファイルのコピー状況に応じてデータセットを作成し直した上で学習を行う必要があるため、新たに追加したループ（6~9 行目）内で呼び出される。各メソッドの詳細は次節で述べる。

アプリケーション内でステージインを行うためには、ファイルのコピー元（外部ストレージ）とコピー先（ローカルストレージ）のディレクトリのパスの情報が必要となる。提案手法では入力引数を拡張してこれらの情報をプログラムに渡しており、したがって提案手法を利用するエンドユーザは入力引数の拡張作業も行う必要がある。

従来の深層学習プログラムでは、Make_Dataset の処理 (リスト 3 の 2 行目) の中で、学習に使用するファイルの順序が決定され、同処理内で学習に使用するデータセット形式に変更される。リスト 4 の 2 行目では、その処理の一部をそのまま利用することで、コピーするファイル名とコピーする順番を管理するためのリスト file_list を、コピー元のディレクトリの情報から作成している。このリストの先頭のファイルからコピーすることで、学習をより早く開始することができる。なお、 3 行目のコピーに関する処理と、4~16 行目の学習に関する処理はそれぞれ別のプロセスとして実行することで、コピーと学習の同時実行を実現している。

なお、リスト 4 の 10~14 行目で示したように、学習を数エポック繰り返す場合は、1 エポック目の学習が完了した時点で学習に使用する全データのコピーが完了しているので、2 エポック目以降は従来の学習と同様に学習する。その際呼び出す Make_Dataset も、従来のプログラムと同様のメソッドを使用する。

リスト 3 従来の深層学習プログラムの全体像

```
Input: data_dir:学習に使用するデータが置いてあるディレクトリ、
    all_epoch:エポック数
 1: Make_Model
 2: Make_Dataset
 3: for i = 1 to all_epoch do
 4:     for j = 1 to all_step do
 5:         Train step(j)
 6:     end for
 7: end for
```

リスト 4 提案手法におけるプログラムの全体像

```
Input: external_dir:コピー元のディレクトリ、local_dir:コピー先
    のディレクトリ、all_epoch:エポック数
 1: Make_Model
 2: make file_list(external_dir)
 3: Copy(file_list from external_dir to local_dir)
 4: for i = 1 to all_epoch do
 5:     if i == 1 then
 6:         while all_step 番目の学習が完了するまで do
 7:             Make_Dataset
 8:             Training
 9:         end while
10:     else
11:         Make_Dataset
12:         for j = 1 to all_step do
13:             Train step(j)
14:         end for
15:     end if
16: end for
```

4.2 Copy

Copy メソッドの処理をリスト 5 に、Copy メソッド内で作成される Copy_Process の処理をリスト 6 に示す。Copy メソッドでは、まず、コピーの完了状況を管理するための配列 copy_check[] を共有メモリ上に作成する（リスト 5 の 1 行目）。上記の配列はコピーするファイルの数と同じ要素

リスト 5 Copy

Input: process_num:コピーを実行するプロセスの数、file_num:コピーするファイル数、file_list:コピーする全ファイルのリスト
1: copy_check = Array(i,file_num)
2: **for** $i = 0$ to process_num **do**
3: make copy_file[] from file_list
4: Copy_Process(copy_file[],copy_check[])
5: **end for**

リスト 6 Copy_Process

Input: copy_file[]:各プロセスに割り当てられたファイルのリスト、copy_check[]:コピー状況を保存する配列
1: **for** $i = 0$ to len(copy_file[]) **do**
2: copy(copy_file[i])
3: change(copy_check[],copy_file[i])
4: **end for**

リスト 7 Make_Dataset

Input: data_dir:データをコピーするディレクトリ、skip_data:データをスキップする場合のスキップ数
Output: dataset:学習に使用する tf.data.Dataset、check_step:何ステップまで学習できるか
1: **if** 必要なデータがコピー完了しているか確認 **then**
2: データセット作成に進む
3: **else**
4: コピー完了まで待機
5: **end if**
6: コピー状況から check_step を計算
7: **if** skip_data **then**
8: data_dir 内のファイルを元に dataset を作成
9: dataset = dataset.skip(skip_data):skip_data で指定した分のデータをスキップ)
10: **else**
11: data_dir 内のファイルを元に dataset を作成
12: **end if**
13: **return** dataset,check_step

数からなる int 型の配列であり、各要素は対応するファイルのコピーが完了しているか否かを表す。上記の配列は生成時にすべての要素が 0（コピーが完了していない状態）に初期化され、Copy_Process によるファイルのコピーが完了すると対応する要素の値を 1（コピーが完了した状態）に変更する（リスト 6 の 3 行目)。前述のように Copy_Process はメインプロセスとは別プロセスで実行されるが、共有メモリ上に配置することで両方のプロセスが上記の配列を参照できるようにした。

一方、Copy_Process メソッドでは、入力引数として渡されたリスト copy_file の情報を元に、リストの先頭から順にファイルのコピーを行う（リスト 6 の 1~4 行目)。copy_file は各プロセスがコピーを担当するファイルのリストであり、file_list の情報を元に Copy メソッドが生成する（リスト 5 の 3 行目)。例えば 10 個のファイルを 2 つのプロセスを用いてコピーする場合、1 つ目のプロセスでは 1、3、5、7、9 番目のファイルを、2 つ目のプロセスでは 2、4、6、8、10 番目のファイルを割り当てることで、file_list の順序とほぼ同じ順序でコピーが完了することが期待できる。

4.3 Make_Dataset

Make_Dataset メソッドの処理をリスト 7 に示す。Make_Dataset では、訓練データファイルから実際に学習で使用するデータセット形式である tf.data.Dataset を作成し、dataset として返す。Make_Dataset では、1 行目で示したように、データセットを作成する前にまずファイルのコピー状況を確認する。Make_Dataset が最初に呼び出されたときは、学習開始に必要なファイル（具体的には file_list の先頭のファイル）のコピーが完了しているか否かを確認する。Make_Dataset が 2 回目以降に呼び出されたときは、次のデータセット作成に必要なファイル（具体的には file_list 上で前回データセット作成時に使用した末尾のファイル以降のファイル）のコピーが完了しているか否かを確認する。データセット作成に必要なファイルのコピーが完了していない場合は、コピーが完了するまで待機する（4 行目)。

Make_Dataset では、データセット作成の他に、作成したデータセットを用いて学習可能なステップ数を表す check_step を計算する。深層学習では一般にミニバッチ学習が行われるが、ミニバッチ学習では 1 つのミニバッチに対する学習をステップという。check_step を Make_Dataset の返り値に追加することで、後述する Training で check_step で指定されたステップ数分の学習を行う。

Make_Dataset は data_dir 内のコピーが完了したファイルと、入力引数として与えられた skip_data を用いてデータセットを作成する。skip_data はこれまでに作成したデータセットに含まれるデータ数を表す変数である。これまでに作成したデータセットに含まれるデータを除外してデータセットを作成することにより、Training では学習を中断したステップの次のステップからの学習を再開できる。

4.4 Training

Training メソッドの処理をリスト 8 に示す。Trainig の入力引数には、Make_Dataset の返り値である dataset と check_step に加え、学習を開始するステップを表す start_step が渡される。これは、従来とは異なり、提案手法では学習を途中のステップから再開する処理が必要なためである。Training は、現在のデータセットで学習可能なステップまでの学習が終わると一旦学習を中断し、次のデー

リスト 8 Training

Input: dataset:学習に使用するデータセット、check_step:何ステップまで学習できるか、start_step:何ステップから学習を開始するか
Output: restart_step:何ステップから学習を再開するか

```
for i = start_step to check_step do
    Train step(i)
end for
return restart_step = check_step + 1
```

表 2: TSUBAME3.0 のシステム構成

項目	詳細
総論理演算性能	12.15 PFLOPS
総ノード数	540
総主記憶容量	135TB
ネットワーク	Intel Omni-Path, Full-bisection Fat Tree

タセットが作成された際はその次のステップから学習を再開する。このために、次に何ステップ目から学習を再開するかを表す restart_step を Training は返す。Training の呼び出し元に返された restart_step の値が、次に Training を呼び出す際の start_step の値となる。

このように提案手法ではデータセットの作成と学習の中断/再開を繰り返すことで、外部ストレージまたはローカルストレージにすべてのファイルが存在する場合と同じ順序で訓練データに対する学習を行う。

5. 評価方法

5.1 実験環境

提案手法の評価を東工大が運用するスーパーコンピュータ TSUBAME3.0 上で行った。TSUBAME3.0 のシステム構成を**表 2** に、ノード構成を**表 3** に示す。表 2 で示したように、TSUBAME3.0 は計算ノード上のローカルストレージ領域の他に、各計算ノードから直接アクセス可能な高速・大容量の外部ストレージ領域を備えており、ファイルステージングを活用した今回の実験に適している。

上記の環境上で TensorFlow を使用し、3 つのニューラルネットワーク (自作した CNN、AlexNet[6]、ResNet50[4]) の学習を行うアプリケーションを作成した。なお、今回使用した TensorFlow のバージョンは TensorFlow2.5.0 であり、訓練データファイルは ILSVRC の 2012 年大会で使われた ILSVRC2012 データセットの学習用データ [2](データサイズ: 147.5GB 、画像枚数:約 120 万枚) を元に作成した。

5.2 実験方法

まず、ファイル I/O が低負荷な状態においてバッチサイズを 128、エポック数を 1、訓練データファイルの総サイズを 147.5GB（ILSVRC2021 のデータセット全体）として、3 つのニューラルネットワークの学習を行った。その際、全てのモデルに対して、ファイルステージングを行わずに（外部ストレージから訓練データファイルを直接読み出しながら）実行した場合、従来のステージング方式を用いて（訓練データファイルをローカルストレージにコピーした後でローカルストレージ上のファイルを読み出しながら）実行した場合、提案手法を用いた場合の 3 つの方式を用いてアプリケーションを実行し、学習時間を計測した。

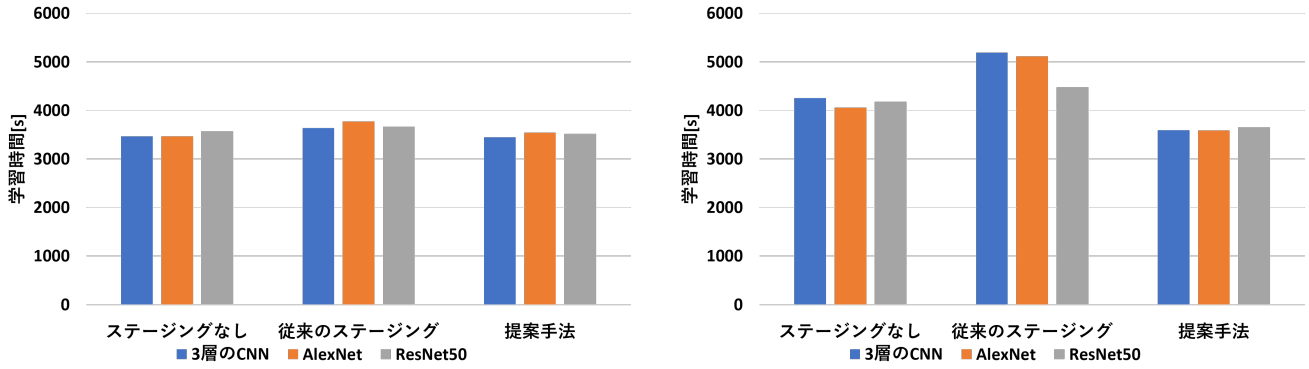
次に、ファイル I/O が高負荷な状態において、3 つのニューラルネットワークの学習を行った。その際、低負荷状態の計測と同じように 3 つの方式を用いて実験を行ったが、ステージングを行わない場合に関しては、バッチサイズを 128、エポック数を 1 とし、訓練データファイルの総サイズが 71.8GB（データセット全体の約 1/2)、147.5GB（データセット全体)、295.1GB（データセット全体の 2 倍）の場合について評価した。従来のステージング方式と提案手法を用いた場合に関しては、バッチサイズを 128、エポック数を 1 または 10 とし、訓練データファイルの総サイズが 71.8GB（データセット全体の約 1/2)、147.5GB（データセット全体)、295.1GB（データセット全体の 2 倍）の場合について評価した。なお上記の実験は、I/O インテンシブなプログラムである IOR を深層学習プログラムを実行する計算ノードとは別の計算ノードで実行することにより、意図的にファイル I/O が高負荷な状態を作り出すことで行った。

表 3: TSUBAME3.0 のノード構成

項目		詳細
CPU	プロセッサ名	Intel Xeon E5-2680 V4 x2
	物理 (論理) コア数	14 (28)
	動作周波数	2.4GHz
	理論演算性能	425.6GFLOPS
Memory	容量	256GB
	メモリ帯域幅	154GB/s
GPU	プロセッサ名	NVIDIA Tesla P100 x4
	コア数	3,584
	メモリ容量	16GB
	メモリ帯域幅	720GB/s
	理論演算性能	5.3TFLOPS（倍精度）
	CPU-GPU 間接続	PCI Express 3.0 (x16)
	GPU 間接続	NVLink (40GBx4)

表 4: 提案手法による実行時間削減率 [%]

	3 層の CNN	AlexNet	ResNet50
ステージングなし	15.6	11.7	12.4
従来のステージング	30.8	29.9	18.3



(a) 低負荷時　　(b) 高負荷時

図 2: 各方式による深層学習ジョブの実行時間

6. 評価結果

3 つの方式による深層学習ジョブの実行時間を**図 2** に示す。この評価では、訓練データファイルの総サイズを 147.5GB、エポック数を 1、バッチサイズを 128 とした。図 2 の (a) と (b) を比較すると、いずれの方式においても、ファイル I/O が高負荷時は低負荷時と比較してジョブの実行時間は増加する。ただし、提案手法に関しては他の方式と比べて実行時間の増分が小さいことがわかる。この結果により、提案手法によって I/O 競合時の学習時間の増加を抑制できていることが確認できた。

ファイル I/O が高負荷時における提案手法の実行時間削減率を**表 4** にまとめる。表 4 によると、実行時間削減率にばらつきがあることが分かる。これは、本実験中に他のユーザによって実行されたジョブの状況によって、I/O 競合時における外部ストレージのアクセス時間がばらついたためである。

また、図 2 によると、低負荷時における各モデルの実行時間にほとんど差がない。これは今回の評価に使用した深層学習プログラムの最適化が不十分なためである。今回 3 つのモデルを実装するにあたり、学習に使用するデータを前処理する部分に関して性能チューニングをほとんど行っていない。TensorFlow にはあるステップの学習中に次のステップで使用するデータを用意する関数が用意されているが、その関数も今回は使用していない。そのため本実験で使用したプログラムでは、各ステップにおいて訓練データファイルへのアクセス、データの前処理、学習がすべて逐次的に行われる。本実験で使用したプログラムは訓練データファイルへのアクセス時間とデータの前処理時間が支配的だったために、モデルの違いによる実行時間の差が生じなかったのだと考えている。

高負荷時において異なるサイズの訓練データファイルを用いて学習を行った場合の実行時間を**図 3** に示す。この評価では、3 層の CNN を使用し、エポック数を 1、バッチサイズを 128 とした。図 3 により、いずれの訓練データファイルにおいても、提案手法によって I/O 競合時の深層学習ジョブの実行時間の増加を抑制できていることがわかる。

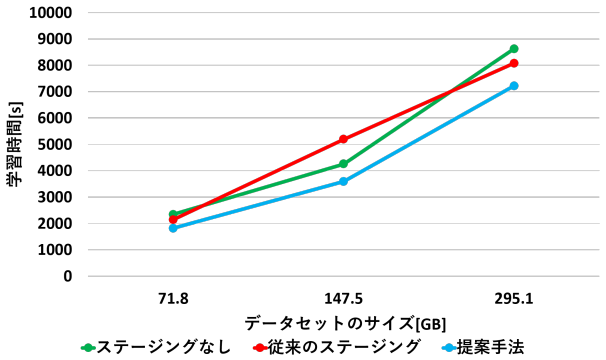


図 3: 高負荷時における訓練データファイルの総サイズと実行時間の関係 (3 層の CNN)

表 5: 提案手法による実行時間削減率 [%]

	71.8[GB]	147.5[GB]	295.1[GB]
ステージングなし	22.3	15.6	16.3
従来のステージング	15.0	30.8	10.6

提案手法による実行時間削減率を**表 5** にまとめる。表 5 によると、実行時間削減率と、訓練データファイルの総サイズや実行方式に関連性は認められない。訓練データファイルを外部ストレージからローカルストレージへ転送する時間と学習に要する時間は、どちらも学習に使用する訓練データファイルの総サイズに比例する。そのため、訓練データファイルの総サイズだけを変更しても、実行時間全体の中でファイルの転送に要する時間の割合はあまり変わらない。そのため、訓練データファイルの総サイズを変更しても提案手法による効果が変わらないのだと考えている。

高負荷時における、エポック数と従来のステージング方式に対する実行時間削減率を**表 6** にまとめる。この評価では、訓練データファイルの総サイズを 147.5[GB]、バッチサイズを 128 とした。表 6 により、いずれのモデルを使用

表 6: 提案手法による実行時間削減率 [%]

	エポック数:1	エポック数:10
3 層の CNN	30.8	1.9
AlexNet	29.9	1.8
ResNet50	18.2	1.2

した場合でも、エポック数が大きいほど、従来のステージング方式に対する提案手法の効果が小さくなっていることが分かる。これは、エポック数が増加するにつれて、深層学習ジョブ全体の実行時間に占めるファイル転送に要する時間の割合が小さくなるからである。その結果、ステージング方式の違いによるジョブの実行時間の差も小さくなっている。

7. おわりに

7.1 まとめ

従来のファイルステージング手法は、I/O 競合が発生してファイルアクセス性能が低下した状況において、深層学習アプリケーションのような全体の実行時間に占めるファイルアクセス時間の割合が大きいアプリケーションに対してあまり効果的でない。

そこで本稿では、アプリケーションの実行と並行してステージインを行う手法を提案した。ステージングを行わない場合、従来のステージング方式を用いた場合、提案手法を用いた場合の 3 つの方式で深層学習ジョブの実行時間を比較した結果、提案手法によりステージインに要する時間が隠蔽され、I/O 競合時の実行時間を最大で 30.8%削減できることが確認できた。

7.2 今後の展望

実行時間全体に占めるステージイン以外の処理時間の割合が小さくなるほど提案手法の効果は大きくなる。特に、今回の実装では学習に使用するデータの前処理を高速化する工夫を行っていないため、ジョブの実行時間が大きく伸びてしまっている。そこで今後はデータの前処理方法の最適化を行い、ジョブの実行時間の短縮を図る予定である。また今回の実験ではノード内の複数 GPU を用いた並列化を行っているが、ノードをまたいでさらに多くの GPU を用いて並列化を行うことで、学習時間をさらに短縮できると考えている。

謝辞 本研究は東京工業大学のスパコン TSUBAME3.0 を用いて行った。

参考文献

[1] 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 (2016).

[2] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K. and Fei-Fei, L.: "Imagenet: A large-scale hierarchical image database", *2009 IEEE conference on computer vision and pattern recognition*, Ieee, pp. 248–255 ((2009)).

[3] Google: Cloud TPU での ResNet のトレーニング — Cloud TPU — Google Cloud, `https://cloud.google.com/tpu/docs/tutorials/resnet?hl=ja`. (Accessed on 01/18/2020).

[4] He, K., Zhang, X., Ren, S. and Sun, J.: "Deep residual learning for image recognition", *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778 ((2016)).

[5] Hori, A., Kamoshida, Y., Matsuba, H., Ohta, K., Yasui, T., Sumimoto, S. and Ishikawa, Y.: On-Demand File Staging System for Linux Clusters, *Proceedings of the ACM International Conference on Supercomputing (ICS'19)*, pp. 296–307 (2009).

[6] Krizhevsky, A., Sutskever, I. and Hinton, G. E.: "Imagenet classification with deep convolutional neural networks", *Advances in neural information processing systems*, pp. 1097–1105 ((2012)).

[7] MathWorks: ディープラーニング – これだけは知っておきたい 3 つのこと - MATLAB & Simulink, `https://jp.mathworks.com/discovery/deep-learning.html`. (Accessed on 10/26/2021).

[8] 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，情報処理学会研究報告，Vol. 2019-HPC-168, No. 25, pp. 1–11 (2019).

[9] 芹沢和洋，建部修見：大規模機械学習訓練における I/O 性能の高速化，情報処理学会研究報告，Vol. 2019-HPC-170, No. 9, pp. 1–12 (2019).

[10] 人工知能学会監修，神嶌敏弘編，麻生英樹・安田宗樹・前田新一・岡野原大輔・岡谷貴之・久保陽太郎・ボレガラダヌシカ共著："深層学習"，近代科学社 書籍 ((2015)).