Vol.2017-HPC-160 No.9
2017/7/26

動的なプロセス数操作による分散深層学習の耐故障性と性能評価

辻 陽平[1,a)]　野村 哲弘[1]　實本 英之[1]　佐藤 育郎[2]　松岡 聡[1]

概要：深層学習はその認識精度の高さから研究開発が盛んに行われており，実社会においても深層学習を取り入れた応用技術を目にすることができる．深層学習では十分な認識精度を得るまでに，大量のデータと GPU などを用いた長時間の計算が必要となる．そのため HPC クラスタなどの高性能計算機での分散処理が利用される．分散システムでは故障発生間隔が短くなる傾向があり，アプリケーションの計算を正しく継続させるために耐故障性の手法が必要になる．
本研究では大規模システム上の深層学習において重要になる耐故障性に対して，既存の checkpoint/restart でない新たな手法 detect/respawn を提案し，これを ULFM-MPI によって実装した．SPRINT と呼ばれる分散深層学習アプリケーションを用いて TSUBAME-KFC の 16 ノード（128GPU）上で提案手法と既存手法を比較したところ，10 時間の学習において提案手法が 2.5%低いエラー率となり，より高い認識精度を達成することができた．

1. 背景

1.1 分散深層学習

近年，深層学習と呼ばれる機械学習の手法が画像認識 [1]，音声認識 [2]，自然言語処理などの分野において他の機械学習手法に比べ高い学習性能を達成している．深層学習ではニューラルネットワークと呼ばれるモデルを用いており，他の機械学習手法に比べモデルパラメータ数が多いといった性質がある [3]．この性質のため深層学習では過学習や計算コストが問題になることが多い．

深層学習ではモデルパラメータ $\mathbf{w}$ と学習用データ $\mathcal{D} = \{d\}$ を用いて誤差関数 $\sum E(\mathbf{w}; d)$ の値を計算し，その値が最小となるよう $\mathbf{w}$ を順次更新していく．更新における手法として最も一般的に用いられるのが確率的勾配降下法（SGD; Stochastic Gradient Descent）である．確率的勾配勾配法ではミニバッチサイズと呼ばれる学習用データ数ごとに $\sum E(\mathbf{w}; d)$ の計算，$\mathbf{w}$ の更新を行う．具体的には式 (1) のように更新する．

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \gamma \sum_{d \in \mathcal{D}_{\mathrm{MB}}^{(t)}} \left(\frac{\partial E(\mathbf{w^{(t)}}; d)}{\partial \mathbf{w}} \right) \qquad (1)$$

ただし，t は更新回数，γ は学習率と呼ばれるハイパーパラメータ，$\mathcal{D}_{\mathrm{MB}}^{(t)}$ は更新回数が t であるときのデータ数がミニバッチサイズと等しい $\mathcal{D}$ の部分集合である．更新されたモデルパラメータ $\mathbf{w}$ の認識精度の評価では，検証用データと呼ばれる学習用データとは別のデータによって行われる．学習用データでは誤差関数と勾配が計算されるが，検証用データでは誤差関数（またはそれが整形された値）のみを計算する．検証用データから計算された値が学習を進めるにつれて小さくなっていれば学習が進んだと判断する．

深層学習では，十分な認識精度を得るまでに大量のデータと長時間の計算が要求される．また認識精度に影響を与えるハイパーパラメータ等が多く存在しそれらの画一的な決定方法がないため最適なハイパーパラメータの探索をしつつ繰り返し学習を行い，認識精度を高めることが一般的である．このように空間的，時間的な計算コストの大きさから分散処理によって高速化を図ることが多い．深層学習における分散化では大きく，データ並列とモデル並列，と呼ばれる 2 種類の分散方法がある．データ並列では各プロセスが同じモデルパラメータ $\mathbf{w}$ のコピーを持ち，異なるデータに対して $\mathbf{w}$ を学習させる．通信はプロセス間で $\mathbf{w}$ を同期させる際に発生する．一方モデル並列ではモデルパラメータ $\mathbf{w}$ そのものを複数のプロセス間で分散化させて持ち，通信によって計算結果を伝搬させる．データ並列とモデル並列のどちらが良いかは使用するニューラルネットワークの形に依存し，また実装の難易度も変化する．本研究で利用した深層学習の実装である SPRINT はデータ並列で分散化している．

[1] 東京工業大学
[2] デンソーアイティーラボラトリ
[a)] tsuji.y.ae@m.titech.ac.jp

1.2 HPC クラスタでの耐故障性

時間的・空間的な計算量が大きい多くの計算は分散処理を用いることで高速化が可能である．科学技術計算に限らず，大量のデータを高速に処理するために今後さらに HPC クラスタやクラウドを用いた分散処理の要求が高まると予想される．分散処理を適用する場合，通常は計算がスケールするよう実装を行う．したがって，プロセス数（またはノード数）が増加するにしたがって計算時間は短縮させることが期待される．プロセス数の増加で計算時間の短縮が期待できるが，一方でプロセス数を増加によって故障の問題が顕著になってくる．定量的にも，N ノードのシステムでは平均故障間隔（MTBF; Mean Time Between Failure）が $1/N$ になることが，幾つかの仮定の下で示すことができる．このように故障が頻繁に発生する場合，計算を正しく継続させるため，そしてハードウェアリソースを無駄にしないため，耐故障性の手法が必要となる．

耐故障性の手法として一般的に利用されるのが checkpoint/restart（以下 C/R と記述）である．C/R では計算の途中経過を定期的に保存し（checkpoint と呼ぶ），故障が発生した際には最新の途中経過から計算を再開する（restart と呼ぶ）．保存先の記憶領域としては特定のノードの二次記憶領域や，共有ファイルシステムを用いることが多い．いずれにおいても記憶領域の故障は無視できるものとしている．

HPC の分野での分散処理のノード間通信制御は MPI（Message Passing Interface）が利用される．MPI は通信インターフェイスの規格であるため，利用する際には Open MPI，MPICH，MVAPICH 等の実装を用いることとなる．MPI は多くの分散計算で利用されている，一方で現在の最新標準規格ではプロセス故障の扱いが定まっておらず [4]，故障したプロセスを除外するなどの復旧処理を行うことができない．そのため，MPI で通信方法を記述しているプログラムで故障が発生した場合，プログラム全体を異常終了させることが一般的である．

2. 分散深層学習での耐故障性

深層学習はその計算コストの大きさから分散処理によって高速化を図ることがある．大規模な分散処理では故障が頻繁に発生するため，耐故障性の手法が重要になってくる．現在，多くの分散深層学習フレームワークは耐故障性の手法として C/R を採用している [5]．しかし，今後のモデルパラメータの巨大化やそれに伴う学習用データの巨大化によって，これまで問題にならなかった C/R のコストが顕著になってくる．具体的には以下のようなものが挙げられる．

- 故障時にシステム全体を停止し checkpoint から再開する際に失われる計算はノード数に比例するため，大規模システムでは失われる計算が非常に多くなってしまう．
- 非同期的に学習を行っている場合，checkpoint を取るときにプログラムを何らかの形で同期させる必要がでてきてしまう．
- モデルパラメータ数が大きい場合に一回の checkpoint にかかる I/O の時間が全体の実行に比べて無視できないものになってしまう．
- HPC クラスタでジョブスケジューラなどが複数のユーザのアプリケーションの実行順序などを管理している場合，C/R（とくに自動で restart しないもの）では再開時にキューに戻され，大きく時間を無駄にしてしまう．

また HPC クラスタで一般的に利用される MPI がプロセス故障を扱えない問題もある．

本研究では C/R に代わる大規模分散システムでの深層学習の計算に適した耐故障性の手法として detect/respawn（以下 D/R と記述する）を提案する．さらにこれを MPI の拡張実装である ULFM（User Level Fault Mitigation）[6] を用いることで深層学習アプリケーション SPRINT 上に実装した．

D/R の概略図を**図 2** に示す．D/R では MPI コミュニケータの故障を検知した場合（detect）にコミュニケータを動的に縮小し，リソースの許す限り，新たなプロセスをコミュニケータに追加し計算を継続する．C/R（**図 1**）との大きな違いはプロセスを動的に除去，追加している点である．これによって 1 つのノードの故障であったとしてもシステム全体の停止することなく対処することが可能である．

本研究では故障という言葉をノード単体で発生し，故障が発生したプロセスが正常な計算を継続することが不可能となり，またその故障を他のノードが検知できるような故障のみに対して用いる．

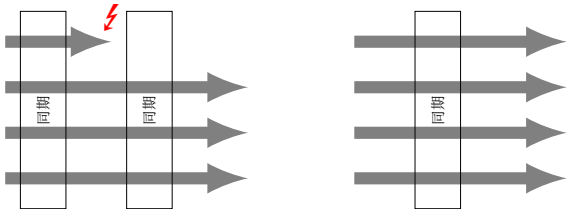

図 1: C/R での復旧方法の模式図．故障の発生を検知した場合，全プロセスを停止させ，再び同じプロセス数で最新の checkpoint から計算を再開する．

3. 実装

3.1 SPRINT

本研究の深層学習の実装として SPRINT を利用した．SPRINT は株式会社デンソーおよびデンソーアイティーラボラトリが開発した分散深層学習アプリケーションであ

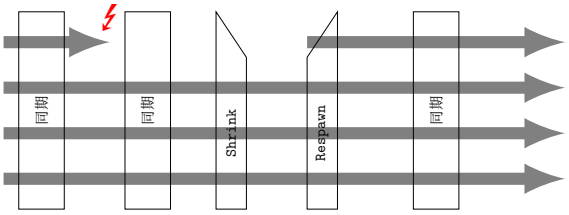

図 2: D/R（提案手法）での復旧方法の模式図．故障の発生を検知した場合，全プロセスを停止させずに，その故障プロセスのみを計算から除外し，新たなプロセスを追加する．

る．SPRINT では計算機 1 ノードに対して，1 つの MPI プロセスが実行され，データ並列による分散学習を行う．各 MPI プロセスではノード内にある GPU のデバイス数に応じて，`pthread_create()` により複数のスレッドが生成される（GPU スレッドと呼ぶ）．MPI プロセスは主にノード間通信を制御し，GPU スレッドは主に学習用データの取得と，勾配計算を行う．**図 3** に SPRINT 実行時のハードウェア構成を示す．

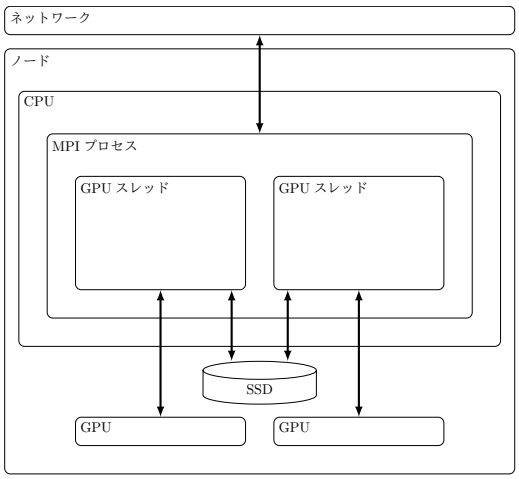

図 3: SPRINT でのハードウェアと MPI プロセス，GPU スレッドの関係図

学習用データは重複のないように各 MPI プロセスの担当の部分が実行時に決定される．モデルパラメータ $\mathbf{w}$ の更新は全プロセスが計算した差分（式 (1) の右辺第二項の総和の部分）が必要になるため，通信を行い全ノードが全ノードの計算した勾配を持てるようにする．SPRNIT ではこの部分を加算 `MPI_Allreduce()` によって実装している．Algorithm 1 に MPI プロセスの疑似コードを示す．更新の操作は `while` 文によって繰り返され，毎イテレーション `MPI_Allreduce()` を実行する．1 ノードにおいて，GPU スレッド同士，そして GPU スレッドと MPI プロセスは非同期的に動作する．

3.2 耐故障性のための実装

3.2.1 Checkpoint/restart

SPRINT では学習の進捗状況を確認するために一定時間

Algorithm 1 MPI プロセスの疑似コード

```
1: ...
2: メモリ空間の確保
3: GPU スレッドの作成，実行開始
4: メモリ空間の値の初期化
5: ...
6: while ¬ 学習終了 do
7:     mutex                          ▷ GPU スレッドとの排他制御
8:         SendBuf ← [w]_n + Δw
9:     end mutex
10:    RecvBuf ← MPI_Allreduce(SendBuf)
11:    mutex                          ▷ GPU スレッドとの排他制御
12:        w ← RecvBuf + μ(RecvBuf − w_old)
                                      ▷ 第二項はモーメンタム計算
13:        w_old ← RecvBuf
14:    end mutex
15: end while
16: ...
```

間隔でモデルパラメータ $\mathbf{w}$（と制御に必要な幾つかの変数値）を共有ファイルシステムに保存している．C/R の実装ではこれを用いた．実装の制限から保存を行うプロセスを固定しているが，本研究ではそのプロセスには故障が発生しないという仮定をおいて実験をしている．また時間間隔はコンパイル時に指定できるため，実験条件によってその都度変更している．

C/R での故障時の挙動はデフォルトのものに従った（特別な実装はしていない）．すなわち，MPI の関数がエラーを検知した場合にプログラム全体を異常終了（abort）する．これによって再開時はハードウェアの再割り当て，メモリの初期化等が必要となる．

故障からの復旧は保存されたモデルパラメータを読み込むことで実現する．SPRINT には学習済モデルパラメータを読み込む機能が備わっているので特別な実装をせずに可能である．

3.2.2 Detect/respawn

本研究の提案手法である D/R は ULFM を用いて実装される．ULFM は MPI Fault Tolerance Working Group によって開発が進められている，ユーザーが MPI でのプロセス故障を管理できるようにした，MPI の拡張規格またはその実装である．現在 ULFM によって提供される関数群や定数群は MPI 標準規格には含まれておらず，利用する場合は開発用の実装を利用することとなる．Open MPI と MPICH において利用が可能である．

D/R での故障の検知は `MPI_Allreduce()` の返り値を判別することで実現する．ULFM も含め MPI で提供される関数群は返り値によってエラーの有無とその種類を判別することができる．正しく動作した場合，関数は `MPI_SUCCESS` という定数を返す．エラーがあった場合は（Open MPI などの）実装が許す限り，その種類を定数によって返す．実装では `MPIX_ERR_PROC_FAILED`, `MPIX_ERR_REVOKED`, `MPI_ERR_PENDING` の 3 種類の定数である場合のみプロセス

復旧を試み，これら以外の場合はプログラムを異常終了する．

プロセス復旧の簡単な流れを図 4 に示す．故障が発生した場合，ULFM が提供する MPIX_Comm_shrink() によってコミュニケータを縮小する．縮小したコミュニケータでは新たにランクが割り当てられる．続いて，縮小したコミュニケータから MPI2 の標準である MPI_Comm_spawn() によって新しいプロセスが作成される．ここで作成されるプロセスは元と同じプログラムであるが，MPI の機能により元のプロセスか新しく作成されたプロセスかどうかの判別は可能である．作成されたプロセスと作成を行ったプロセスは互いに別々のコミュニケータに属するため，コミュニケータをマージして 1 つのコミュニケータにする．この際，ランクを故障前と同じになるように揃える操作を行う．最後にここまで学習済のモデルパラメータ **w** やその他必要な変数値を作成したプロセスに送信し，復旧が完了する．これ以降は通常の計算を継続することができる．

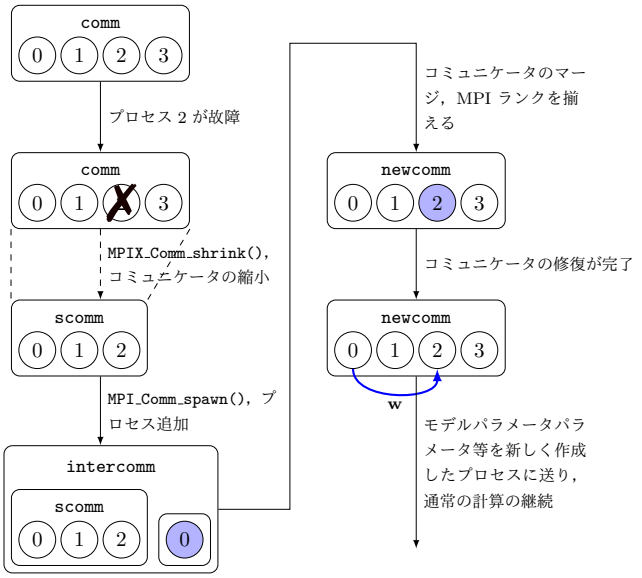

図 4: D/R での故障発生から復旧までの流れ

3.3 Detect/shrink

多くの HPC クラスタでは様々なユーザのアプリケーションを限られたハードウェアのリソース上で効率良く実行するためにジョブスケジューラが実行されている．ジョブスケジューラを利用しているシステムではアプリケーションの実行前にハードウェアリソースを指定するため，実行中に追加のハードウェアを要求するのは困難である場合がある．故障発生時にプロセスを動的に追加するのではなく，取り除き，追加を行わずに計算を継続させるものも実装した（Detect/shrink）．これは図 4 において MPIX_Comm_shrink() を呼ぶ箇所で操作を止め，その後プロセスが減少した分の計算を再配分することで実装可能である．

4. 評価

4.1 実行環境と前提条件

実装の実行環境として，東京工業大学，学術国際情報センターの TSUBAME-KFC を利用した．TSUBAME-KFC では 1 ノードに対して CPU が 2 つ，GPU が 8 つ搭載されている．表 1 に TSUBAME-KFC の実験時のハードウェア構成とソフトウェア構成を示す．

表 1: TSUBAME-KFC の実行環境

CPU	Intel Xeon CPU E5-2620 v2 × 2
周波数	2.1 GHz
物理コア数/CPU	6
L3 キャッシュ/CPU	15 MB
メモリ/ノード	64 GB
メモリハンド幅/ノード	42.6 GB/s
グラフィックカード	NVIDIA(R) Tesla(R) K80 × 4
GPU/カード	NVIDIA(R) Tesla(R) GK 210 × 2
周波数/GPU	最大 875 MHz
メモリ/カード	GDDR5 24 GB
メモリバンド幅/カード	480 GB/s
インターコネクト	Mellanox(R) Infiniband 4X FDR
バンド幅	56 Gbps
インターフェイス	PCI Express Gen 3 × 8
OS	CentOS Linux release 7.3.1611
ジョブスケジューラ	SLURM 16.05.7
C コンパイラ	g++ (GCC) 4.8.5
MPI	Open MPI 1.7.1 (ULFM 1.1)
CUDA	CUDA 8.0

深層学習における学習対象のネットワークとしては 11 層の畳み込みニューラルネットワーク（CNN; Convolutional Neural Network）を利用した．学習用データは ImageNet [7] の ILSVRC2012 で用いられた画像データセットの一部を用いる．元のデータは 1000 クラスに分類されたデータであるが，実験では 1000 クラス中 32 クラスに該当するデータのみを抜き出し，全 22GB のデータとしている．検証用データは同様に 32 クラスに分類された全 967MB のデータである．いずれのデータも各ノードのローカルな SSD に予め書き込まれており，事前に移動等をする必要はない．

故障発生は，予め定めたプロセスが SIGKILL シグナルをプログラム内で呼ぶことで疑似的に発生させる．これによって故障プロセスが応答しない状況を作り出す．

評価を行うにあたって幾つかの仮定をおく．

- 実験に使用する最大ノード数が 16 であるので，1 回の学習における複数ノードの故障が非常に発生しにくい．したがって，1 回の学習では 1 回の故障しか発生しないと仮定した．
- C/R，D/R いずれにおいても，故障から復旧は即座に行われる．評価を単純にするため復旧に必要なハード

ウェアリソースは復旧時に必ず確保できるものと仮定した．

4.2 オーバーヘッド

まず，復旧と保存に必要となるオーバーヘッドを計測した結果を示す．

復旧ではプロセスが `SIGKILL` を呼んだ時刻から，再び更新の `while` ループに入る時刻までを計測した．**図 5** に C/R と D/R でそれぞれが復旧までにかかった時間（秒）を示した．2 ノードから 16 ノードまで計測した平均を測定値としている．ノード数によって大きな差は見られなかった．図より，D/R が C/R と比べて少ない時間で復旧が可能であることがわかる．

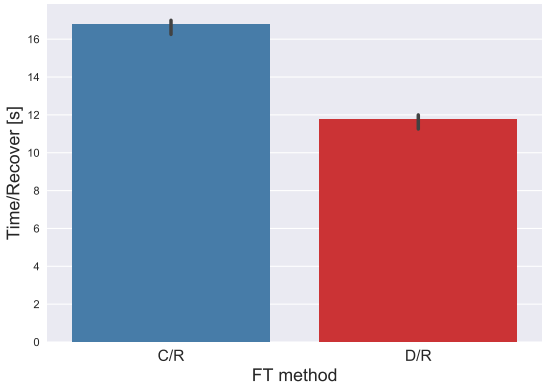

図 5: C/R（左）と D/R（右）における復旧にかかった時間（秒）．ノード数による大きな変化は見られなかった．

図 6 に 1 回の checkpoint にかかる時間（秒）を示した．SPRINT での checkpoint は固定された 1 つのノードの 1 つのスレッドのみが非同期的に行っている．いずれのノードにおいても 1 回の checkpoint は 2 秒に満たない時間となった．この値は深層学習での計算が数時間から数日かかることと，保存を数十分に 1 回程度しか行わないことを考慮すると，実行時間に大きな影響を与えることがないと考えられる．**表 2** に checkpoint にかかる時間の関数の簡単な内訳を示す．実行は 16 ノードのものを使用している．表から見てとれるようにほとんどの時間は共有ファイルシステムへの書き込みの時間となっている．実行環境では I/O のバンド幅が大きいため，約 95MB あるモデルパラメータを保存するのに大幅な時間は必要としない．バンド幅が小さい場合や，モデルパラメータのデータサイズが大きい場合，保存のオーバーヘッドは顕著になるため，checkpoint を頻繁に行うことができなくなる．

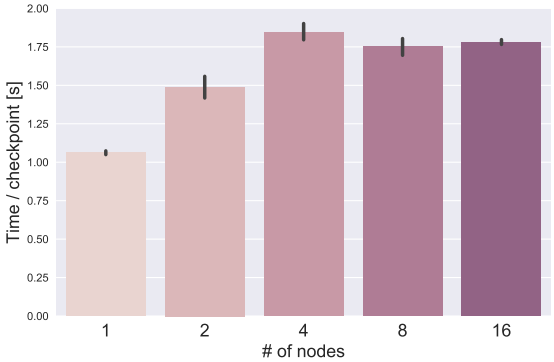

図 6: 1 ノードから 16 ノードにおける 1 回の保存にかかる時間（秒）．

表 2: 1 回の checkpoint にかかる時間の関数の内訳．

	平均時間 [s]	checkpoint との割合 [%]
checkpoint	1.7547	100%
`fwrite()`	1.7222	98%
`cudaMemcpy()`	0.0324	1.8%

4.3 学習曲線

続いて学習がどのように進むか観察する．SPRINT では認識精度の評価のためにモデルパラメータを保存する必要があるため，C/R，D/R いずれにおいても 5 分おきにモデルパラメータを保存するように設定した．これは保存のオーバーヘッドが全体の計算に比べて無視できるものという仮定の下で設定している．C/R では故障によって失われる計算を見るため，復旧時のモデルパラメータの読み込みは 1 時間に 1 回保存した場合のものを読み込んでいる．

図 7 に 16 ノードの，**図 8** に 4 ノードの場合の実行結果を示す．x 軸が時間（分）であり，y 軸がエラー率（認識精度の誤差値，小さな値である方が良い）である．いずれのノード数でも 12 時間学習させ，故障を 2 時間 42 分後に発生させている．そのため C/R ではおよそ 42 分間の計算を失うこととなる． D/R は動的にプロセスを追加可能で

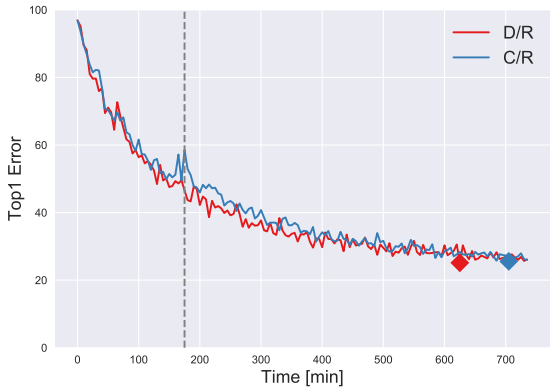

図 7: 16 ノード，128GPU 上で 2 時間 42 分後に故障を発生させた場合の学習曲線．ひし形のマークは 12 時間学習したまでの最小エラー率である．灰色の点線が故障発生時間である．

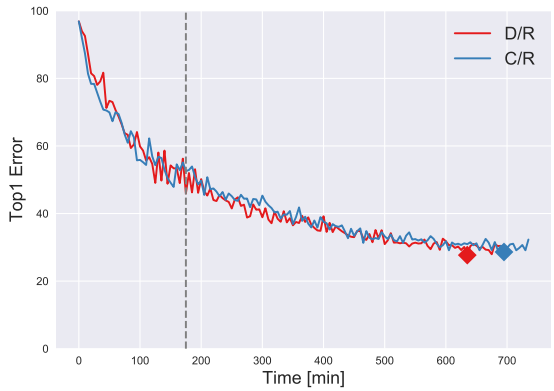

図 8: 4 ノード，16GPU 上で 2 時間 42 分後に故障を発生させた場合の学習曲線．ひし形のマークは 12 時間学習したまでの最小エラー率である．灰色の点線が故障発生時間である．

あるため，C/R で必要であった計算の巻き戻しがないことが確認できる．

表 3 に 10 時間学習時点と 12 時間学習時点での最小エラー率を示す．D/R は失う計算がないため最終的なエラー率は小さいことが分かる．

表 3: D/R と C/R での 10 時間と 12 時間学習後のそれぞれの最小エラー率．

耐故障性手法	10 時間学習後	12 時間学習後
D/R（16 ノード）	27.5%	25.13%
C/R（16 ノード）	30.06%	25.56%
D/R（4 ノード）	31.5%	27.69%
C/R（4 ノード）	31.31%	28.62%

5. 関連研究

5.1 HPC における耐故障性

Gamell ら [8] は既存の C/R の耐故障性手法がエクサスケールの計算に適していないことから Fenix と呼ばれる耐故障性を確保できるようなフレームワークを ULFM を用いて開発した．Fenix では detect/respawn の手法を取っている．すなわち，プロセス故障時には新たなプロセスを `MPI_Comm_spawn()` によって起動し，MPI コミュニケータに追加することで計算の継続を試みる．新しいプロセスはこれまで計算したデータを保持していないので，checkpoint のデータを読み込むことで計算に参加することが可能となる．Fenix では checkpoint を各ノードの隣接したノードのメモリ内に書き込むことで通常発生する I/O アクセスの時間を短縮している．実験では S3D と呼ばれるアプリケーションを 24 時間，Titan Cray-XK7 上で実行し，故障を 47 秒おきに人為的に発生させた．結果として，全計算時間の 30%が耐故障性の獲得に必要な計算の割合となった．これは 1 日あたり 9 回の故障が発生する現在の S3D のアプリケーションと実行と同じ割合であった．さらに Fenix が採用した checkpoint の方法によって 250,000CPU コアの分散システムまでスケールすることが示された．しかしこのフレームワークでは深層学習のデータ並列では必要のない高頻度の checkpoint を行っているため，愚直に分散深層学習に適用することはできない．

Guo ら [9] は HPC クラスタ上での MapReduce の計算における耐故障性のモデルとして C/R ではない detect/resume と呼ばれるものを提案した．Detect/resume モデルでは checkpoint を適切に利用しつつ，故障時に故障プロセス自体を MPI コミュニケータから取り除き，計算を再配分することで継続を試みる．さらに彼らは提案したモデルを既存の MapReduce ライブラリを改変することで実装した．FT-MRMPI と呼ばれるこのライブラリを用いて 256 ノード上の HPC クラスタで実行したところ，計算完了までの時間として，C/R の手法に比べ 10 から 12%減少させることができた．彼らの研究は MapReduce の計算に特化しているため，本研究ではより深層学習に適した耐故障性の手法の開発，実装した．

5.2 分散深層学習のフレームワーク

Fox ら [5] は深層学習フレームワークを幾つかの観点から比較を行った．彼らは分散手法についても比較を行い，耐故障性の有無，そしてその手法等について整理した．TensorFlow [10] や MXNet [11] などに代表されるほとんどの分散深層学習フレームワークは学習済のモデルパラメータを再度学習するため，また最小のエラー率のモデルパラメータを保存するために C/R の機能が備わっている．しかし，いずれのフレームワークも checkpoint と restart はユーザが明示的にプログラムする必要がある．

6. まとめと今後の課題

本研究では代表的な耐故障性の手法である C/R が大規模な分散深層学習の計算に適してないとして，新たな手法である D/R を提案した．HPC クラスタ上での実行を想定し，提案手法を ULFM を用いて SPRINT 上に実装した．TSUBAME-KFC の 16 ノード（128GPU）上で既存手法である C/R と提案手法を比較したところ，同じ学習時間であってもより高い認識精度を達成することができた．

今後の課題として，ジョブスケジューラのリソースマネジメントに対応して，故障時にハードウェアリソースが使える場合は新たに確保し detect/respawn の手法を，できない場合は detect/shrink の手法を使用するようなシステムの開発が必要である．さらに，耐故障性の観点以外でもユーザの要求に応じて，ハードウェアリソースを動的に割り当てる分散深層学習システムの開発をすることが必要である．

謝辞 本研究の一部は、JST CREST(JPMJCR1303, JPMJCR1687) 及び，産総研・東工大実社会ビッグデータ活

用オープンイノベーションラボラトリの支援による．

参考文献

[1] Krizhevsky, A., Sutskever, I. and Hinton, G. E.: Imagenet classification with deep convolutional neural networks, *Advances in Neural Information Processing Systems*, p. 2012.

[2] Amodei, D., Ananthanarayanan, S., Anubhai, R., Bai, J., Battenberg, E., Case, C., Casper, J., Catanzaro, B., Cheng, Q., Chen, G., Chen, J., Chen, J., Chen, Z., Chrzanowski, M., Coates, A., Diamos, G., Ding, K., Du, N., Elsen, E., Engel, J., Fang, W., Fan, L., Fougner, C., Gao, L., Gong, C., Hannun, A., Han, T., Johannes, L., Jiang, B., Ju, C., Jun, B., LeGresley, P., Lin, L., Liu, J., Liu, Y., Li, W., Li, X., Ma, D., Narang, S., Ng, A., Ozair, S., Peng, Y., Prenger, R., Qian, S., Quan, Z., Raiman, J., Rao, V., Satheesh, S., Seetapun, D., Sengupta, S., Srinet, K., Sriram, A., Tang, H., Tang, L., Wang, C., Wang, J., Wang, K., Wang, Y., Wang, Z., Wang, Z., Wu, S., Wei, L., Xiao, B., Xie, W., Xie, Y., Yogatama, D., Yuan, B., Zhan, J. and Zhu, Z.: Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin, *Proceedings of The 33rd International Conference on Machine Learning* (Balcan, M. F. and Weinberger, K. Q., eds.), Proceedings of Machine Learning Research, Vol. 48, New York, New York, USA, PMLR, pp. 173–182 (online), available from ⟨http://proceedings.mlr.press/v48/amodei16.html⟩ (2016).

[3] 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，Bollegala, D.：深層学習 — Deep Learning，近代科学社 (2015).

[4] Forum, M. P. I.: *MPI: a Message-passing Interface Standard: Version 3.1*, High-Performance Computing Center (2015).

[5] Fox, J., Zou, Y. and Qiu, J.: Software Frameworks for Deep Learning at Scale, *Internal Indiana University Technical Report* (2016).

[6] Rizzi, F., Morris, K., Sargsyan, K., Mycek, P., Safta, C., Debusschere, B., LeMaitre, O. and Knio, O.: ULFM-MPI Implementation of a Resilient Task-Based Partial Differential Equations Preconditioner, *Proceedings of the ACM Workshop on Fault-Tolerance for HPC at Extreme Scale*, FTXS '16, New York, NY, USA, ACM, pp. 19–26 (online), DOI: 10.1145/2909428.2909429 (2016).

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

[8] Gamell, M., Katz, D. S., Kolla, H., Chen, J., Klasky, S. and Parashar, M.: Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, Piscataway, NJ, USA, IEEE Press, pp. 895–906 (online), DOI: 10.1109/SC.2014.78 (2014).

[9] Guo, Y., Bland, W., Balaji, P. and Zhou, X.: Fault Tolerant MapReduce-MPI for HPC Clusters, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, ACM, pp. 34:1–34:12 (online), DOI: 10.1145/2807591.2807617 (2015).

[10] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I. J., Harp, A., Irving, G., Isard, M., Jia, Y., Józefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D. G., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P. A., Vanhoucke, V., Vasudevan, V., Viégas, F. B., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y. and Zheng, X.: TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems, *CoRR*, Vol. abs/1603.04467 (online), available from ⟨http://arxiv.org/abs/1603.04467⟩ (2016).

[11] Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C. and Zhang, Z.: MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems, *CoRR*, Vol. abs/1512.01274 (online), available from ⟨http://arxiv.org/abs/1512.01274⟩ (2015).