DEIM Forum 2018 G1-5

ビックデータ分散処理基盤におけるパラメータ制御の一検討

加藤　香澄† 　竹房あつ子†† 　中田　秀基††† 　小口　正人†

† お茶の水女子大学
〒 112–8610 東京都文京区大塚 2–1–1
†† 国立情報学研究所
〒 101–8430 東京都千代田区一ツ橋 2–1–2
††† 産業技術総合研究所
〒 305–8560 茨城県つくば市梅園 1–1–1
E-mail: †{g1320510,oguchi}@is.ocha.ac.jp, ††takefusa@nii.ac.jp, †††hide-nakada@aist.go.jp

あらまし　近年，お年寄りや子供を見守るサービスや防犯カメラなどによるライフログの利用が普及し，多様に活用されるようになってきている．また，ディープラーニング技術が非常に発達してきており，画像認識や音声認識を始めとする様々な分野に応用されている．しかし正確な認識処理を行うためには大量のデータ処理が必要となるため，処理の並列化が求められる．本研究では，ディープラーニングフレームワークである Chainer をクラスタコンピューティングプラットフォーム Apache Spark 上で動作させ，分散並列機械学習処理を行う．実験から，Spark のパラメータ調整による処理の効率化について検討する．

キーワード　分散処理，並列処理，Spark，Chainer

A Study on Parameter Control in Big Data Distributed Processing Infrastructure

Kasumi KATO†, Atsuko TAKEFUSA††, Hidemoto NAKADA†††, and Masato OGUCHI†

† Ochanomizu University
2-1-1 Otsuka,Bunkyo-Ku, Tokyo 112–8610, Japan
†† National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, 101-8430, Japan
††† National Institute of Advanced Industrial Science and Technology (AIST)
1-1-1 Umezono, Tsukuba, Ibaraki 305–8560, Japan
E-mail: †{g1320510,oguchi}@is.ocha.ac.jp, ††takefusa@nii.ac.jp, †††hide-nakada@aist.go.jp

1. はじめに

近年カメラやセンサ等の発達やクラウドコンピューティングの普及により，一般家庭でのライフログの取得とそのデータの蓄積が可能になった．これらの技術は，遠隔地から家庭にいるお年寄りや子供，ペットを見守ることができる安全サービスや，防犯対策・セキュリティといった用途に応用されている．しかし，サーバやストレージを一般家庭に設置して取得・蓄積した動画像データの解析をするのは困難であるため，センサから取得した動画像データはクラウドに送信して解析する必要がある．ここで，動画像はデータサイズが大きいため，クラウドでの機械学習処理による動画像データ解析に要する計算量は大きくなる．また，クラウドには非常に多くの家庭からデータが送信されることが想定されるため，クラウドでの並列機械学習処理が必要不可欠である．

我々は，大規模データ分散処理プラットフォーム Apache Spark(以降，Spark と呼ぶ) [1] を用いて，ディープラーニングフレームワーク Chainer [2] による機械学習処理を並列化することで，動画像データ解析処理の効率化を検討してきた [3]．本稿では，Spark のパラメータ調整による処理性能を調査し，並列処理の効率化について検討する．

2. 関連技術

本研究ではクラスタでの負荷分散の基盤として Spark，動画

像データの解析処理に Chainer を用いる．以下に各ソフトウェアの概要を述べる．

2.1 Apache Spark

Spark は，高速かつ汎用的であることを目的に設計されたクラスタコンピューティングプラットフォームである．カリフォルニア大学バークレー校で開発が開始され，2014 年に Apache Software Foundation に寄贈された．マイクロバッチ処理という極小単位でのバッチ処理を行うことが特徴であり，演算をオンメモリで行うためアプリケーションがメモリ内にデータを保存でき，高コストなディスクアクセスを避けて処理全体の実行速度を向上させることができる．

Spark 上では RDD(Resilient Distributed Dataset) にデータを保持し，用意されているメソッドを用いて操作することで自動的に分散が可能である．この RDD は，Spark コアで定義されている．Spark コアにはタスクスケジューリング，メモリ管理，障害回復，ストレージシステムとのやりとりといった Spark の基本的機能が備わっている．Spark プロジェクトは，Spark コアと構造化データを扱う Spark SQL，ライブストリーム処理を実現する Spark Streaming，一般的な機械学習の機能を含むライブラリである MLlib，グラフ処理を担う GraphX といった複数のコンポーネントが密接に結合して構成されている [4]．

Spark は前述の RDD とメソッドの組み合わせによって繰り返しの機械学習，ストリーミング，複雑なクエリ，そしてバッチなど幅広い領域を簡単に表現できる．Hadoop [5] などの他のビックデータのツールとの組み合わせが可能であり，優れた汎用性を備えている．

2.2 Chainer

Chainer は Preferred Networks が開発したディープラーニングのフレームワークである．Python のライブラリとして提供されており，その特徴として「Flexible(柔軟性)」「Intuitive(直感的)」「Powerful(高機能)」の 3 つを掲げている．

多くのディープラーニングフレームワークは，一度ニューラルネット全体の構造をメモリ上に展開し，その処理を順に見て順伝播・逆伝播を実行するというアプローチを取っている．一方，Chainer は実際に Python のコードを用いて入力配列に何の処理が適用されたかだけを記憶しておき，それを誤差逆伝播の実行に利用する．このアプローチにより，畳み込みやリカレントなどの様々なニューラルネットや複雑化していくディープラーニングにも対応している．

シンプルな記法で直感的にコードを記述できる点や， CUDA をサポートしており GPU による高速演算が可能である点，インストールが簡単である点も大きな特徴である．画像処理，自然言語処理，ロボット制御など幅広い分野に用いられている．

3. 実験概要

本研究では図 1 のようなフレームワークを想定している．各家庭に設置されたセンサやカメラから動画像データがクラウドに送信されると，Spark のマイクロバッチ処理によるストリーミング処理と並列処理をクラウドの Spark クラスタで行い，得られた結果をユーザに返す．実験では，Spark の担う 2 つの処理のうち，並列処理に着目して実験を行い，その効率化を目指す．実験には，0 から 9 の手書き数字の 28 × 28 画素の画像データに正解ラベルが与えられているデータセットである MNIST [6] を用いた．図 2 に示すように，Python のプログラムを実行して，MNIST を Spark に読み込ませて RDD に変換し，ワーカで Chainer を呼び出して RDD を渡し，その評価を行う．実験では，マスタ 1 台とワーカとして最大 5 台の端末を Spark Standalone Mode で接続する．マスタでプログラムが実行され，各ワーカでのタスクが完了してワーカからマスタに結果が返って出力されるまでに要する時間を測定した．

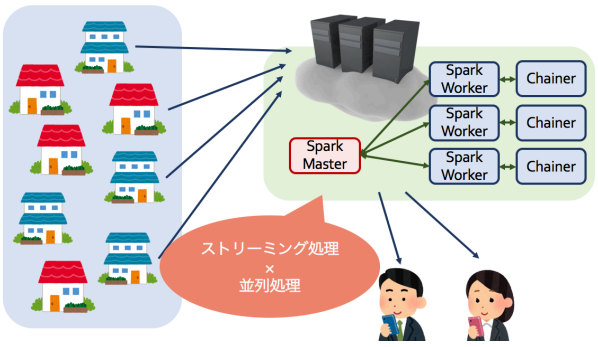

図 1　想定フレームワーク

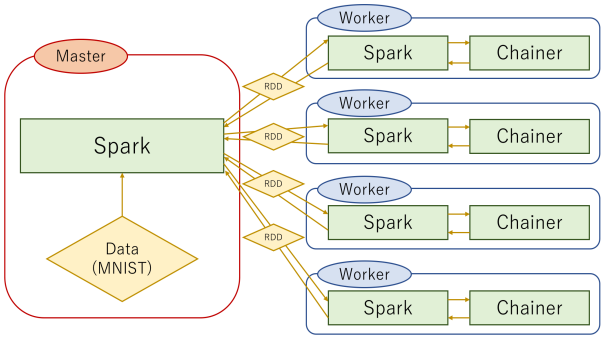

図 2　Spark と Chainer を用いたマスタ・ワーカ処理

本稿では，以下の 2 種類の観点で実験を行った．

実験 1　パーティションの作成方法の比較

実験 2　エグゼキュータ数・コア数の比較

実験 1 では，Spark に読み込ませるデータのパーティションの作成に関するパラメータを以下 3 つの手法で調整し，その性能を調査した．

（1） メソッド partitionBy() を利用

（2） メソッド repartition() を利用

（3） partitionBy() においてパーティショナを調整

partitionBy() と repartition() はどちらとも Spark に備わっているメソッドであり，partitionBy() はパーティション数とパーティショナを，repartition() はパーティション数を引数にとる．手法 3 では partitionBy() の引数としてタスクをラウンドロビン的に分配できるようなパーティショナを与える．実験 2 では，エグゼキュータ数及びそのエグゼキュータのコア数を設定ファイルに記述することで指定した．エグゼキュータとは，Spark が RDD に対して変換やアクションといった操作を行う場所であり，指定なしで実行した場合には 1 ノードに 1 エグゼキュー

表 1　実験で用いた計算機の性能

OS	Ubuntu 16.04LTS
CPU	Intel(R) Xeon(R) CPU W5590 @3.33GHz (4 コア) × 2 ソケット
Memory	8Gbyte

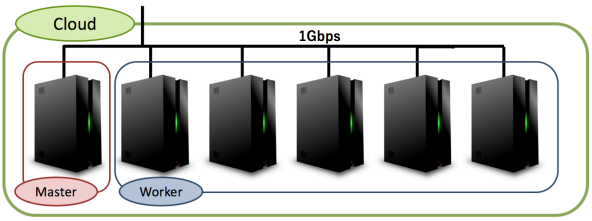

図 3　実 験 環 境

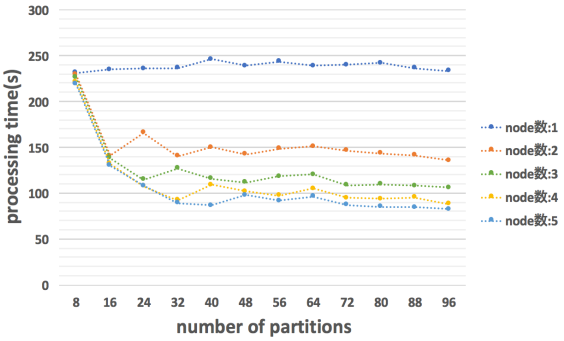

図 4　partitionBy() を用いた際の実行時間の変化

タが作成される．エグゼキュータのコア数の指定なしで実行した場合，そのノードで使用可能なコア数を Spark が自動で取得し、設定する．エグゼキュータ数は spark.executor.instans，コア数は spark.executor.cores を用いることで指定できる．

また，各手法において Spark に読み込ませるデータのパーティション数とワーカのノード数をそれぞれ変化させて測定した．実験で用いた計算機の性能を表 1 に示す．マスタ及び 1〜5 台の全ワーカには同質のノードを用いており，図 3 に示すように 1Gbps のネットワークで接続されたクラスタ構成とした．

4. 実 験 結 果

4.1 実験 1　パーティションの作成方法の比較

1000 個タスクを用意し各設定でノード数を 1〜5，パーティション数を 8〜96 まで 8 刻みで変化させた際の実行時間を 3 回の平均値で示す (図 4，6，8)．また，各設定においてノード数 5，パーティション数 96 におけるタスクごとの処理時間の計測結果を示す (図 5，7，9)．図中の細かな矢印 1 つ 1 つがタスクを示しており，右上方向に連続する矢印の組が各パーティションを示している．

4.1.1　メソッド partitionBy() の利用

実行時間の測定結果を図 4 に示す．実験の結果，ノード数の増加により実行時間が 1/3 ほどまで減少することがわかった．また，パーティション数 32 ほどで実行時間が横ばいになった．

タスクごとの処理時間の測定結果を図 5 に示す．図から，5 つのノード全てにタスクが分配されていることがわかった．しかし，ノードごとに処理の開始時間に差があることや各ノード

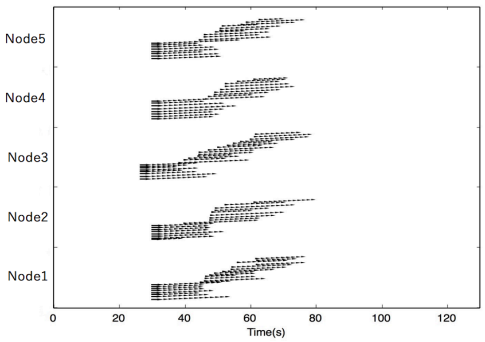

図 5　partitionBy() を用いた際のタスク処理時間

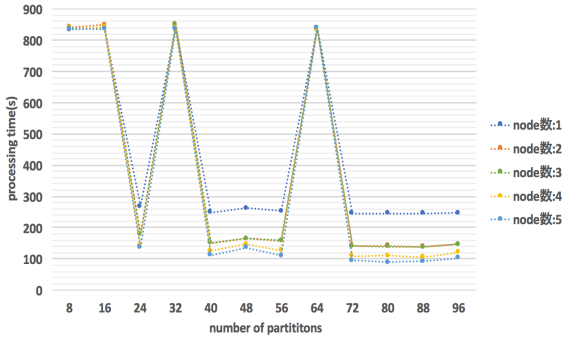

図 6　repartition() を用いた際の実行時間の変化

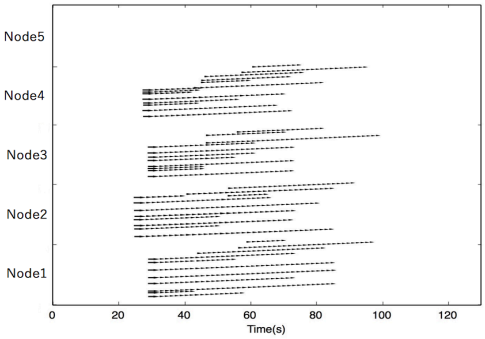

図 7　repartition() を用いた際のタスク処理時間

上において 8 コア並列に使用されいないことから，効率的な分散が行えていないことがわかる．

4.1.2　メソッド repartition() の利用

実行時間の測定結果を図 6 に示す．ノード数増加による実行時間の減少は見られたが，指定パーティション数 8,16,32,64 で実行時間が極端に遅くなっていた．これは，パーティション内のタスク数が 0 であるパーティションが多数できてしまったことによるもので，例として指定パーティション数 32 のときパーティション内のタスク数は 560 が 1 個，440 が 1 個，0 が 30 個になっていた．

タスクごとの処理時間の測定結果を図 7 に示す．図から，ノード 5 つのうち 4 つしか使用されていないことがわかる．また，partitionBy() を利用したときと比較してパーティション内のタスク数にばらつきがあり，多数のタスクを内包するパーティションの処理により，実行時間全体に律速が起きていると考えられる．

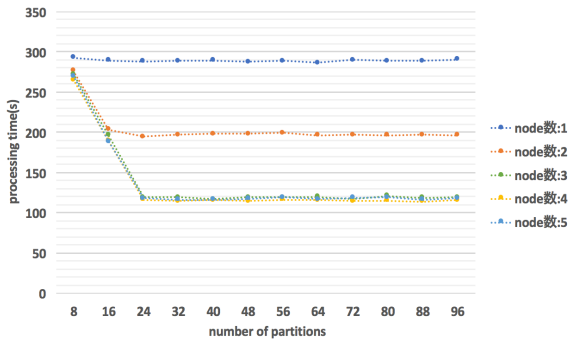

図 8　パーティショナを調整した際の実行時間の変化

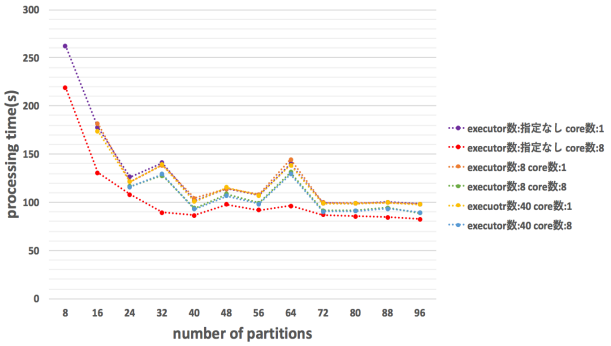

図 10　エグゼキュータ数とコア数調整による実行時間

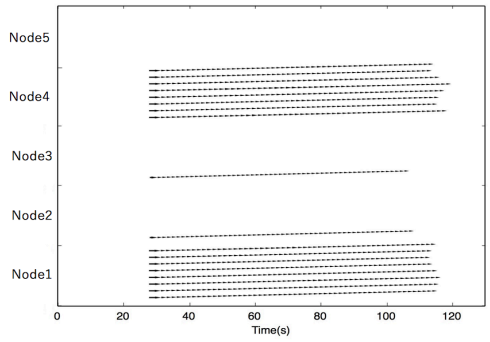

図 9　パーティショナを調整した際のタスク処理時間

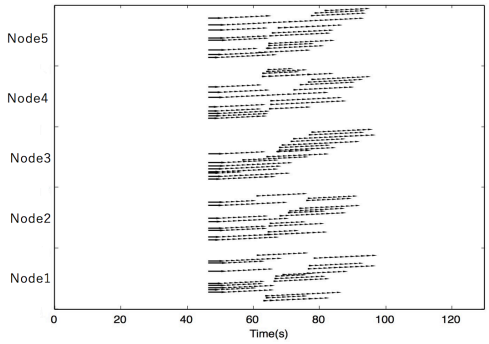

図 11　エグゼキュータ数 8，コア数 1 の指定の場合のタスク処理時間

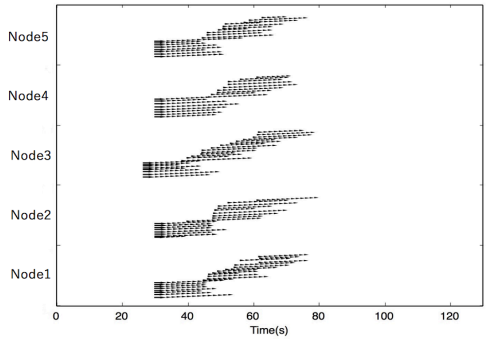

図 12　エグゼキュータ数 8，コア数 8 の指定の場合のタスク処理時間

4.1.3　パーティショナの調整

測定結果を図 8 に示す．図から，ノード数 3 以上で実行時間が一致し，結果が横ばいになることがわかった．また，この調整によりパーティション数の指定に関わらず，パーティション内のタスク数は 18 が 55 個，10 が 1 個とほぼ均等になっていたが，ノード数が増加してもタスクが割り当てられるコアが偏ってしまったため，図のような結果となっていた．

タスクごとの処理時間の測定結果を図 9 に示す．図から，ノード 5 つ中 4 つしか使用されていないことがわかる．処理の開始時間は全ノードでほぼ一致しており，パーティション内のタスク数もほぼ均一になっているように見えるが，並列に動作するコアの数が大幅に減少し，実行時間は遅くなってしまった．

4.2　実験 2　エグゼキュータ数・コア数の比較

1000 個タスクを用意し各設定でノード数を 5 に固定し，パーティション数を 8〜96 まで 8 刻みで変化させた．1 ノードあたりのエグゼキュータ数を指定なし，8，40 と変化させ，コア数は 1，8 と変化させた。計 6 パターンのチューニングで各 3 回ずつ実行時間の計測を行った平均値で示す (図 10)．この実験においてパーティションの作成には partitionBy() を用いている．

図において，エグゼキュータ数 8，コア数 1 の場合とエグゼキュータ数 40，コア数 1 の場合にはパーティション数 8 のとき途中でタスクがロストしてしまい，測定することができなかった．また，エグゼキュータ数 8，コア数 8 の場合とエグゼキュータ数 40，コア数 8 の場合でもパーティション数 8，16 のときにタスクがロストしてしまった．図から，エグゼキュータのコア数を 1 に指定した場合，エグゼキュータ数の指定による挙動の変化はほぼなかった．エグゼキュータのコア数を 1 に指定した場合，エグゼキュータ数 8，40 の計測結果はほぼ一致したが，指定なしの場合は異なる挙動となった．この調整においては，エグゼキュータ数指定なし，コア数 8 の場合に最も処理を速く終了できることを確認できた．

エグゼキュータのコア数 1 の場合と 8 の場合で挙動がどのように変化しているか調査するため，タスクごとの処理時間の計測も行った (図 11，12)．

この 2 つの図から，どちらの場合も 1 ノードで 8 つのパーティションが並列に処理されていることがわかる．また，コア数 8 の場合はパーティションが 8 つスケジューリングされたノードから順次処理が始まっているのに対し，コア数 1 の場合は全てのノードの処理がほぼ同時に始まっている．

5.　理論実行時間の予測

ノード数 5 の場合での理論実行時間の予測を行った (図 13)．上記実験結果から，パーティションの作成方法は partitionBy() を用い，コア数は 8 に指定する．理論実行時間では，パーティション内のタスク数がほぼ均一になり，各ノードのタスク処

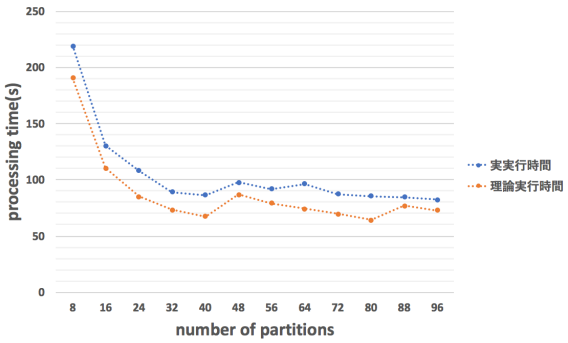

図 13　実実行時間と理論実行時間

理開始時間が最速でそろう状況を仮定している．図 13 から，Spark による分散処理では高速化の余地が残されていることが確認できる．

6. 関 連 研 究

柳瀬らの研究 [8] では，MapReduce [9] [10] モデルに Map と Reduse の反復やデータ型の制限を加えることで，機械学習の並列化に最適化した分散計算プラットフォームが提案されている．評価実験において提案手法の高速性とスケーラビリティが示されているが，膨大なデータ量を扱う際の挙動が課題となっている．

伍らの研究 [11] では，非決定情報システム (NIS) でラフ集合理論の構築を通して，不完全なデータも対象とするデータマイニング手法の研究 [12] における，Spark のクラスタコンピューティング機能を用いた処理の高速化がなされている．

また，MPI を用いた分散処理により Chainer を用いた学習処理を高速化する ChainerMN [13] が Chainer の追加パッケージとして開発された．既存の Chainer の学習コードから数行変更することで，通常の学習に All-Reduce のステップを追加し，全ワーカが求めた勾配を利用して学習の高速化がなされている．

本研究では，一般的なデータ処理インフラストラクチャを用いた機械学習の並列化処理の高速化を目指している．

7. まとめと今後の予定

Chainer によるデータ解析処理を Spark で並列化し，負荷分散を行って性能を調査した．実験 1 でデータのパーティションの作成に関する設定を行い，partitionBy() と repartition() という 2 つのメソッドを利用してその処理の挙動の違いを確認するとともに，partitionBy() におけるパーティショナの調整を試みた．実験結果から，repartition() を利用した場合よりも partitionBy() を利用した場合の方が分散が効率よく為されており，実行時間が短くなることがわかった．また，パーティショナの設定によりパーティション内のタスク数をほぼ均一にすることはできるが，タスクが特定のノードに偏ってしまい，処理の効率化はあまり図れていないことがわかった．

実験 2 ではエグゼキュータ数とエグゼキュータにおけるコア数の設定を行った．実験結果から，エグゼキュータ数指定なし，コア数 8 に指定の場合に処理を速く終了できることがわかった．

以上 2 つの実験結果を元に理論実行時間を予測し，少なくとも 10 秒ほどの改良の余地があることを確認した．

今後の課題として，ログの解析を進めるとともに，調整したパーティショナで切り分けたパーティションを効率よく複数ノードに振り分ける手法を検討する．

謝　　辞

この成果の一部は，JSPS 科研費 JP16K00177，平成 29 年度国立情報学研究所公募型共同研究および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) の委託業務の結果得られたものです．

文　　献

[1] Apache Spark, https://spark.apache.org/.
[2] Tokui, S., Oono, K., Hido, S. and Clayton, J.: Chainer: a Next-Generation Open Source Frame- work for Deep Learning, In Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Informa- tion Processing Systems (NIPS) (2015). 6 pages.
[3] 加藤香澄，竹房あつ子，中田秀基，小口正人: ビッグデータ分散処理基盤を用いた機械学習処理並列化の一考察，マルチメディア，分散，協調とモバイル DICOMO2017 シンポジウム，p. 796-802 (2017).
[4] Karau, H., Konwinski, A., Wendell, P. and Zaharia, M.: Learning Spark, Cambridge: O'Reilly Media (2015).
[5] Apache Hadoop, https://hadoop.apache.org/.
[6] Lecun, Y., Cortes, C. and Burges, C. J.: The MNIST Database of handwritten digits, http://yann.lecun.com/exdb/mnist/.
[7] Chiu, D.-M. and Jain, R.: Analysis of the increase and decrease algorithms for congestion avoidance in computer networks, Computer Networks and ISDN Systems, vol. 17, pp. 1-14 (1989).
[8] Yanase, T., Hiroki, K., Itoh, A. and Yanai, K.: MapReduse Platform for Parallel Machine Learing on Large-scale Dataset, Transactions of the Japanese Society for Artificial Intelligence, vol. 26, No. 5, pp. 621-637 (2011).
[9] Dean, J. and Ghemawat, S.: MapReduse: Simplified Data Processing on Large Clusters, OSDI'4: Sixth Symposium on Operating System Design and Implementation (2004).
[10] Dean, J. and Ghemawat, S.: MapReduse: simplified Data processing on large clusters, Communications of the ACM, vol. 51, No. 1, pp. 107-113 (2008).
[11] Wu , m., Yamaguchi, N., Nakata, M. and Sakai, H.: Improving the Performance of NIS-Apriori Algorithm by Parallel Processing, Proceedings of the Fuzzy System Symposium, vol. 30, pp. 592-595 (2014).
[12] Sakai, H. Okuma, H., Wu M., Nakata, M.: Rough nondeterministic information analysis for uncertain information, The Handbook on Reasoning-Based Intelligent Systems, World Scienti c, pp. 81-118 (2013).
[13] ChainerMN, https://chainermn.readthedocs.io/en/latest/index.html.