大規模データ分散処理プラットフォーム Apache Sparkを用いた分散並列機械学習に関する考察

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が，動画像解析に要する通信量や計算量，プライバシに関する問題が介在している．また，ディープラーニングの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である Apache Spark 上で動作させることによる，分散並列機械学習処理に関して考察する．

キーワード 分散処理，並列処理，機械学習，Spark，Chainer

Consideration on Distributed Parallel Machine Learning using Apache Spark,a Large-scale Data Distributed Processing Platform

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
2-1-1 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なので，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センサ・クラウド間の通信量が膨大になり，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も膨大になる．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，クラウドに多くの負荷がかかってしまう．従ってこの問題については，動画像データの機械学習処理を並列化してクラウドの負荷を軽減する必要がある．

本研究では，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させる

ことで，動画像データ解析処理の効率化を図る．

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 [3]．

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 [4] 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,「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

本稿では，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 [5] を用いて実験を行う．図 1 で示すように，マスタで Python のプログラムを実行し，MNIST を Spark に読み込ませて RDD を作成する．作成した RDD をワーカに渡して，ワーカにて Chainer を用いた MNIST の評価を行う実験を Spark の分散機能を利用して実施した．

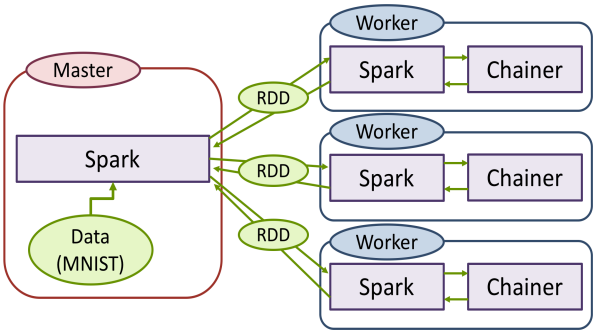

図 1　Spark と Chainer を用いたマスタ・ワーカ処理

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 (8 コア) × 2 ソケット
Memory	8Gbyte

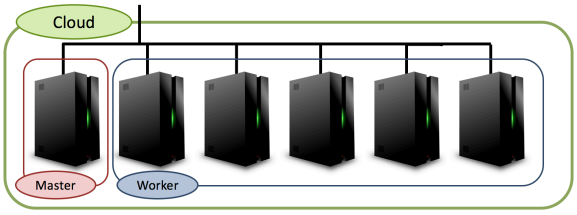

図 2　実 験 環 境

3.1 実験概要

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．本実験では，以下 2 つのパラメータを変えて測定した．

（1） Spark に読み込ませるデータの partition 数

（2） ワーカのノード数

また，この実験を元にタスクがどのように各ノードに分配されているのかを観測する実験や，ジョブの割り当てタイミング，割り当てられたジョブの終了時間を観測する実験を行った．

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 2 に示すクラスタ構成とした．

3.2 実験結果

3.2.1 実行時間

測定結果を図 3 に示す．このグラフは，ノード数を 1〜5 まで変化させ，partition 数を指定なし,2,3,5,10,20,30,40,50 と変化させた際の測定結果 10 回の平均値を用いて作成している．グラフの一番左，partition 数の指定がない場合には Spark の機能に

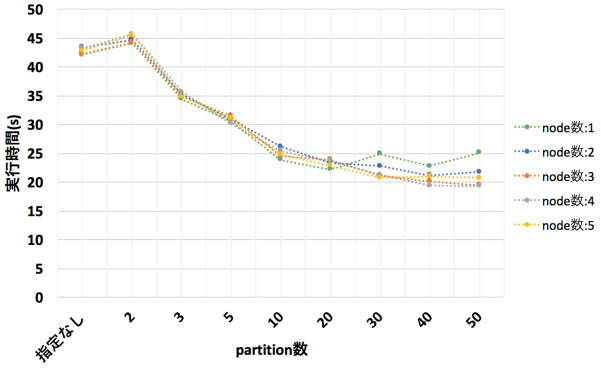

図 3　partition 数及びノード数の変化による実行時間

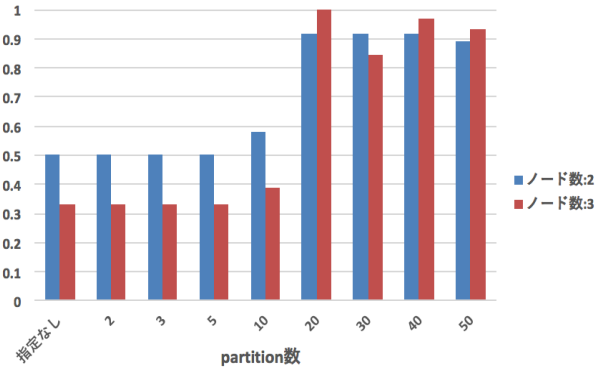

図 4　Fairness Index

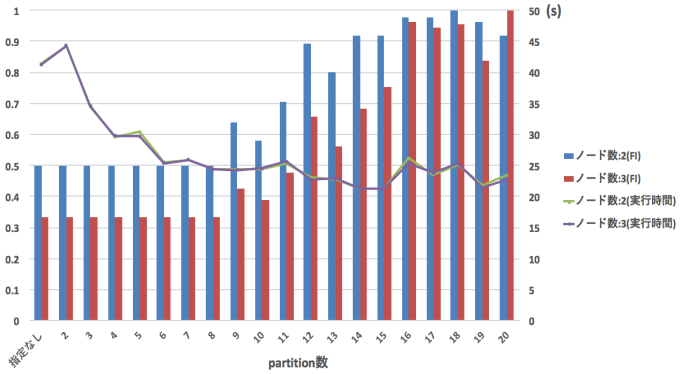

図 5　Fairness Index と実行時間変化

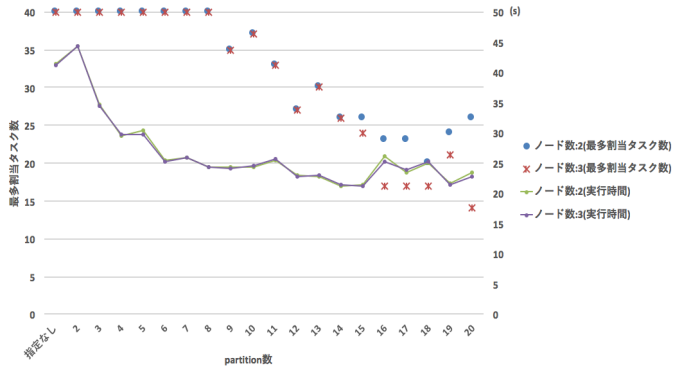

図 6　最多割り当てタスク数と実行時間変化

より partition 数に 2 が自動で設定される．また，partition 数の指定には Spark に用意されているメソッドである partitionBy() を用いている．

実験の結果，partition 数が増加すると実行時間が約 1/2 ほどまで減少することがわかった．また，partition 数の増加による実行時間の減少は partition 数 40 ほどで横ばいになった．一方，ノード数の増加による実行時間の減少はわずかであり，効率よく分散処理が行えていないことがわかった．partition 数 2 のときに一番実行時間がかかっている原因として，partition 数をユーザが指定するとデータを切り分ける際に余分に時間がかかり，指定しない場合よりも遅くなってしまうためと考えられる．

3.2.2　タスク分配

図 4 に，ノード数が 2 と 3 の場合についてタスクがどのように各ノードに分配されているのかを観測した結果を，公平性を示す指標である Fairness Index [6] で示す．Fairness Index は以下の式で計算でき，値が 1 に近いほど公平性が高いことを示す．

$$FairnessIndex : f_i = \frac{(\sum_{i=1}^{k} x_i)^2}{k(\sum_{i=1}^{k} x_i^2)}$$

結果から，実行時間に減少が見られた場合でも実際にはタスクが偏って分配されてしまっていたことが判明した．グラフより，partition 数 10 から 20 の間で公平性に大きな変化が見られた．

図 5 に，partition 数 20 まで細かく計測した際の Fairness Index 及び実行時間変化を図 5 に示す．

より正確な数値を計算するため，30 回ずつ測定を行った結果を用いて Fairness Index を計算した．図 5 から，ノード数 2 と 3 の場合両方について実行時間の変化はほぼ一致し，partition 数 9 以上から複数ノードに渡るタスクの分配が行われていることがわかった．また，公平性が非常に高くなっている partition 数 16〜20 について公平性と実行時間は必ずしも反比例にならなかった．

ノードに割り当てられた最多割り当てタスク数及び実行時間変化を図 6 に示す．実験では全 40 のタスクをノードに分配している．図 6 から，partition 数 8 までは 1 つのノードで全てのタスクが処理されていることがわかった．ノードに割り当てられた最多タスク数はノード数が 2 と 3 の場合で partition 数 14 まで一致していた．また，partition 数 15〜20 において最多割り当てタスク数の増加に応じた実行時間の増加や減少が見られなかったため，最も多くタスクが分配されているノードのタスク処理に要する時間による律速の影響はほとんど無いと考えられる．

3.2.3　ジョブ割り当てタイミング

次に，ノードごとにどのようなタイミングでジョブが与えられているのか調査した．特に，partition 数指定なし及び partition 数 2〜3 と partition 数 16〜20 の 2 つの区間に着目した．

上記の 2 つの区間について，ノード数を 3 として計測した結果を図 7 と図 8 に示す．前述のとおり，partition 数の指定がない場合にはデフォルトで partiton 数が 2 に設定されるが，ここで partition 数の指定がない場合と partition 数 2 の場合で最初のジョブ割り当て時間に差が生じるのは，partitionBy() メソッ

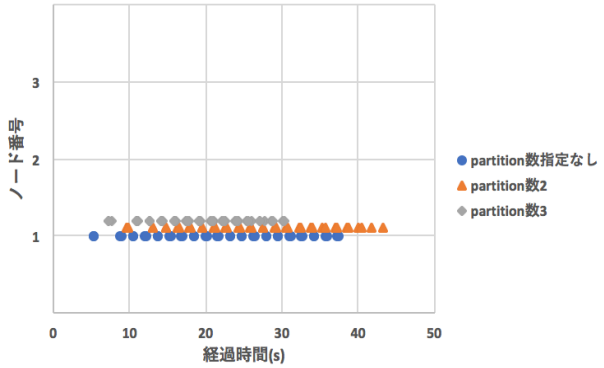

図 7　ジョブ割り当てタイミング (partition 数 指定なし，2，3)

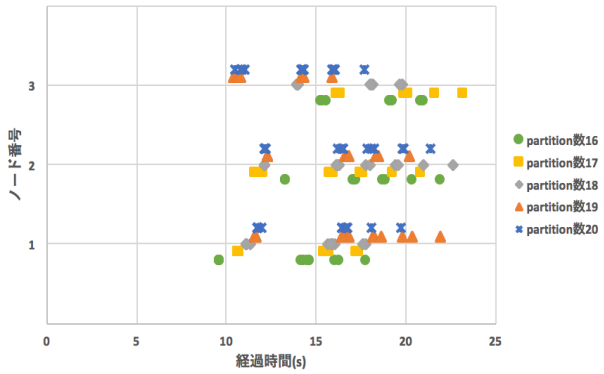

図 8　ジョブ割り当てタイミング (partition 数 16〜20)

ドで partition を設定するのに時間を要するためだと考えられる．これにより図 7 では partition 数 2 のジョブ投入開始時間は partition 数指定なしのジョブ投入開始時間より遅くなっている．partition 数 3 の場合，分散処理による実行時間の減少の影響がメソッドの利用による実行時間増加の影響を上回り，処理全体として実行時間の削除に成功している．

図 8 では，partition 数 16〜20 になるとジョブ割り当てのタイミングに大きな違いは現れなかったが，最初のジョブ割り当ての時間に注目すると partition 数が多いほど割り当てまでにわずかに時間を要することが伺え，その時間は 10 秒前後であった．

以上のことから，ある程度の実行時間減少・タスク分配の公平性をもって本研究における分散並列処理を行うには，RDD 作成・partition 設定などに関して 10 秒前後の時間を要することがわかった．また，効率的な分散処理を行えた際の実行時間の目安として

$$\frac{(\text{初期状態の実行時間}) - (RDD\ \text{作成時間等})}{\text{ノード数}}$$

が推測できる．

3.2.4　タスク処理時間

ジョブ割り当てのタイミングに加え，処理の終了時間にも着目して再び計測を行った．特に，partiton16〜19 に着目して計測した結果を図 9〜図 12 に示す．

図 9〜図 12 から，各ノードに割り当てられる partition の先頭タスクの処理時間はノード内で一致しており，各 partition 内の先頭のタスクの処理に約 3 秒，2 つめ以降に約 1.4 秒ほどか

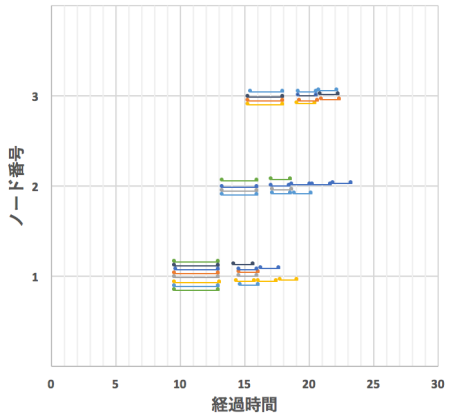

図 9　partition 数 16 の割り当てタスク処理時間

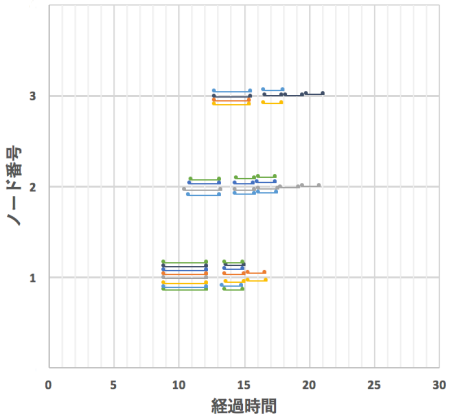

図 10　partition 数 17 の割り当てタスク処理時間

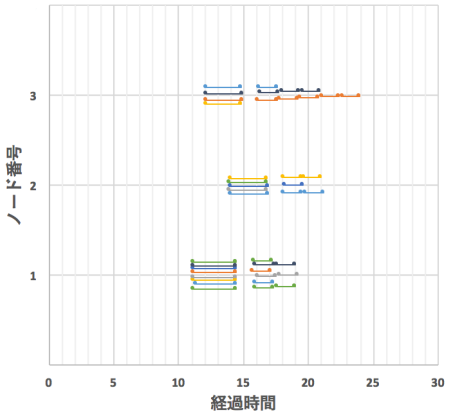

図 11　partition 数 18 の割り当てタスク処理時間

かることがわかる．また，先頭タスクを処理してから 2 つめのタスクの処理を始めるまでの時間より，2 つめのタスクを処理してから 3 つめのタスクの処理を始めるまでの時間の方が短くなっている．図 6 の実行時間を考慮すると，時間がかかってしまっている partition 数 16，18 のときにはある 1 つの partition にタスクが多く分けられてしまっており，その処理時間による律速の影響で全体の実行時間が遅くなってしまっていた．

各 partition 数について複数回割り当てタスクの処理時間を計測したうちの partition 数 17 の 2 回の結果を以下の図 13 と

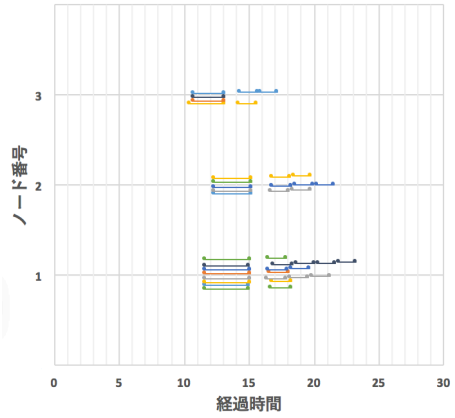

図 12　partition 数 19 の割り当てタスク処理時間

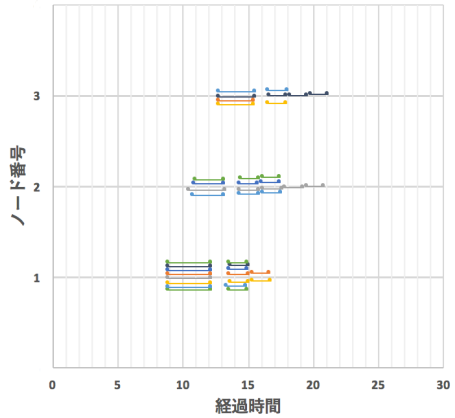

図 13　partition 数 17 の割り当てタスク処理時間 (22.971(s))

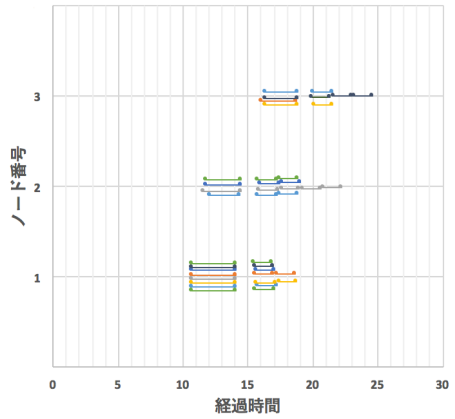

図 14　partition 数 17 の割り当てタスク処理時間 (26.383(s))

図 14 に示す.

図 13 が実行時間 22.971 秒，図 14 が実行時間 26.383 秒のときの計測結果である．partition の分けられ方や，各 partition 内のタスク数は両結果で一致している．先頭ジョブの割り当てられるタイミングが右のグラフのほうが左のグラフより 2 秒ほど遅くなっていることに加え，3 番ノードに割り当てられたジョブの処理時間全体が遅くなっていることにより，実行時間の差が生じていることがわかる．

以上の結果から，タスク割り当ての最適化を図ることで，処理性能の向上が見込めることが明らかになった．

4. 関 連 研 究

柳瀬ら [7] の研究では，MapReduse [8] [9] 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 [10] の研究では，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 [11] 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．

本研究では機械学習の並列化処理のさらなる高速化を目指している．

5. まとめと今後の予定

Chainer による解析処理を Spark で並列化し，負荷分散を行った．実行時間とタスクとジョブの割り当てに関して実験し，ノード数増加による実行時間の減少はわずかであること，タスク割り当ての公平性と実行時間が反比例しないことに加え，ある程度実行時間を減少させてノード間の公平性を保ちつつ処理を行うためにはワーカでの評価を開始する前に 10 秒ほどの処理時間を要することがわかり，本研究における並列処理の目安となる実行時間が推測できた．さらに，ノードに割り当てられた partition とその partition 内のタスクの処理時間を測定することにより，タスクを多く分けられた partition の処理に要する時間の律速の影響で実行時間が遅くなってしまうことがわかった．

今後の課題として，各 partition に分けられるタスクをできるだけ均等にする手法を検討すると同時に，振る舞いの実態をより詳しく調査していくことにより現時点での並列処理の課題を明らかにし，ノード数増加による実行時間の減少を目標とした処理の効率化を図る．

謝　　辞

この成果の一部は，JSPS 科研費 JP16K00177 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] Karau, H., Konwinski, A., Wendell, P. and Zaharia, M.: Learning Spark, Cambridge: O'Reilly Media (2015).
[4] Apache Hadoop, https://hadoop.apache.org/.
[5] Lecun, Y., Cortes, C. and Burges, C. J.: The MNIST Database of handwritten digits, http://yann.lecun.com/exdb/mnist/.
[6] Chiu, D.-M. and Jain, R.: Analysis of the increase and decrease algorithms for congestion avoidance in computer net-

works, Computer Networks and ISDN Systems, vol. 17, pp. 1-14 (1989).

[7] 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).

[8] Dean, J. and Ghemawat, S.: MapReduse: Simplified Data Processing on Large Clusters, OSDI'4: Sixth Symposium on Operating System Design and Implementation (2004).

[9] Dean, J. and Ghemawat, S.: MapReduse: simplified Data processing on large clusters, Communications of the ACM, vol. 51, No. 1, pp. 107-113 (2008).

[10] 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).

[11] 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).