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

加藤 香澄[1]　竹房あつ子[2]　中田 秀基[3]　小口 正人[1]

概要：各種センサーや防犯カメラなどによるライフログの収集，利用が可能となり，お年寄りや子供を見守るサービスなど多様に活用されるようになってきているが，動画像の収集・解析に要する通信量や計算量が課題となる．また，近年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．しかし複数カメラから送信される大量の動画像の解析を高速に行うには，処理の並列化が求められる．そこで本研究では，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おいて分散の挙動を調査する実験を行い，その実態と高速化について考察する．

A Consideration on Parallel Machine Learning Proccessing using a Distributed Big Data Proccessing Infrastructure

Kasumi Kato[1]　Atsuko Takefusa[2]　Hidemoto Nakada[3]　Masato Oguchi[1]

1. はじめに

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ため，センサから取得した動画像データをクラウドに送信して解析する必要がある．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，クラウドでの処理時間が長くなってしまう．従って，クラウドでの動画像データ処理を並列化して高速化を図る必要がある．

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[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の並列化を図る．Spark による分散の挙動を詳細に調査する実験を行い，その結果から処理を効率化・高速化する手法について考察する．

[1] お茶水女子大学
[2] 国立情報学研究所
[3] 産業技術総合研究所

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可能である．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し，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 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 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．利用者が，Spark コアで定義されている 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処理を記述することで自動的に分散処理される．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 社が開発したディープラーニング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 つの特徴を掲げている．

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．

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，その処理を順に実行して順伝播・逆伝播を行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．

3. 実験

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用いることでクラウドの Spark クラスタにおいて解析処理を分散させ，得られた結果をユーザに返す．実験においては 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．

3.1 実験概要

本稿では，0 から 9 の手書き数字の 28 × 28 画素の画像

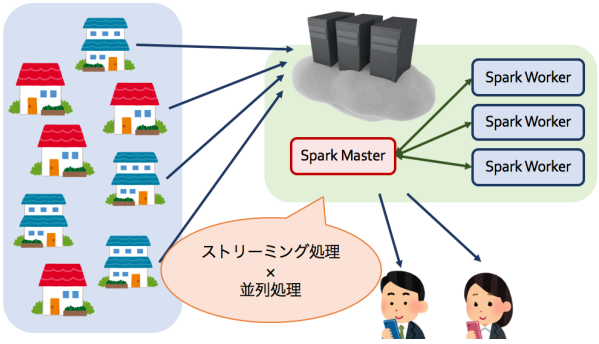

図 1 想定フレームワーク

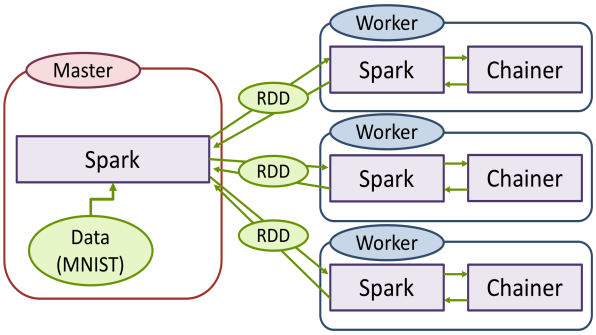

図 2 Spark と Chainer を用いたマスタ・ワーカ処理

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 2 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．

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．また，実行時間の測定結果を元にタスクの各ノードへの分配状況と，割り当てられた各タスクの処理時間を調査した．

本実験では，以下 3 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
(3) locality wait 時間

partition 数は Spark に用意されているメソッドである partitionBy() を用いて指定して変化させることができる．ノード数は Spark の Standalone Mode で接続するワーカの台数を増減させることで変化させる．Spark には Fairness と Locality の両立を達成するためのスケジューリングが実装されている [12]．この考えは Delay Scheduling と呼ばれており，スケジューリングの公平性とデータのローカリティにおける矛盾を防ぐために用いられる．Fairness の観点から次にスケジューリングされるべきジョブがローカルタス

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

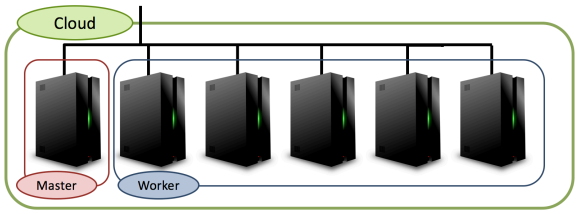

図 3　実験環境

クに着手できないときはそのジョブを待機させ，代わりに他のジョブにタスクに着手させる．本稿ではこの待機時間のことを簡略化して locality wait 時間と呼び，configuration ファイルに記述することで Fairness と Locality の比重を変化させて実験する．

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 に示すクラスタ構成とした．また，各ワーカノードには 1 つずつエグゼキュータがあり，そのスレッド数は 8 になっている．

3.2　実験結果

3.2.1　実行時間

測定結果を図 4 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 1，5，10，15... と 5 刻みで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．このとき，locality wait 時間はデフォルトの 3 秒とした．図 4 から，partition 数が増加すると実行時間が約 1/3 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 25 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．

実行時間が大幅に減少している partition 数 1〜25 に関して再び計測を行った結果を図 5 に示す．図 5 からも，ノード数増加による実行時間の減少がわずかであることが伺えた．また，実行時間は partition 数 8 までは減少し続けるが，それ以降は一定の範囲で増減することがわかった．

partition 数 1〜25 に関して locaity wait 時間を 0 秒，1 秒として実行時間の計測を行った結果をそれぞれ図 6，図 7 に示す．図 6 と図 7 から，locality wait 時間を変更しても実行時間に大きな変化が現れず，ノード数 1 のときに比べてノード数 2 以上のときの実行時間は短くなるが，ノード数をそれ以上増やしても実行時間のさらなる減少は見受けられなかった．

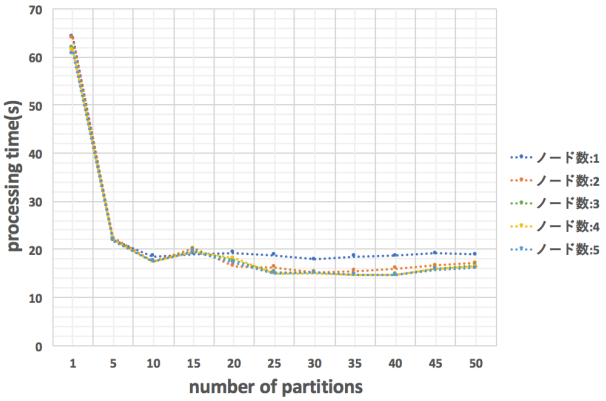

図 4　partition 数及びノード数の変化による実行時間

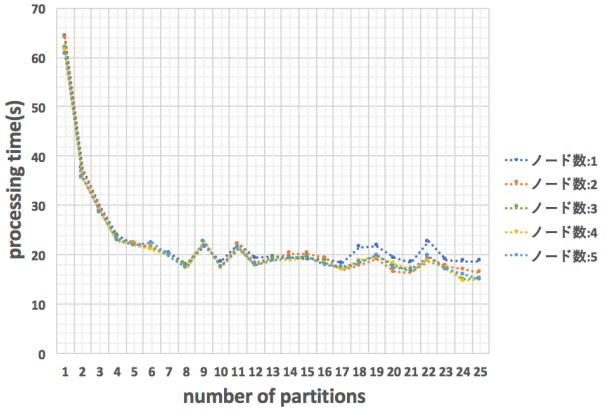

図 5　partition 数及びノード数の変化による実行時間 (partiton 数 1〜25)

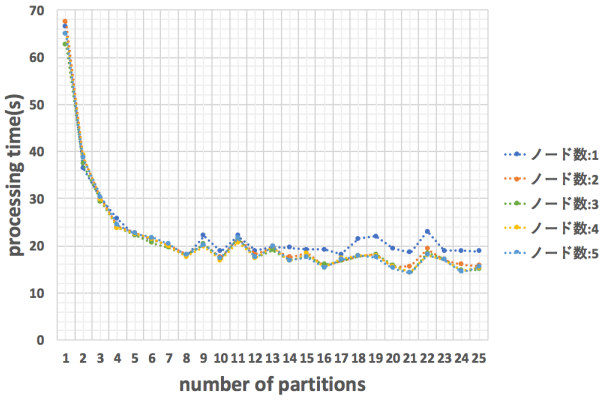

図 6　wait 0 秒の実行時間 (partiton 数 1〜25)

3.2.2　タスク分配状況

図 8 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．

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

図 8 は locality wait 時間をデフォルトの 3 秒の設定で計測したものである．結果から，実行時間に減少が見られた

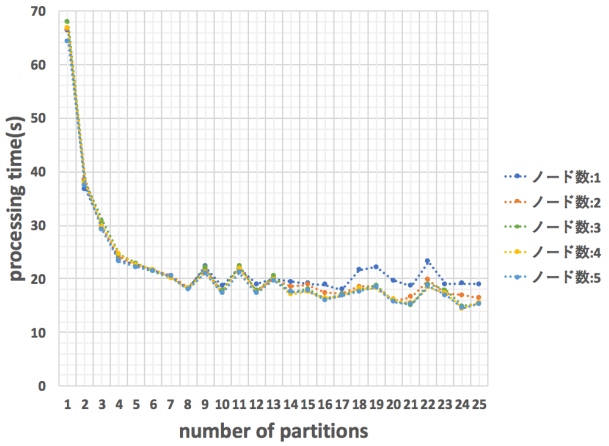

図 7 wait 1 秒の実行時間 (partiton 数 1〜25)

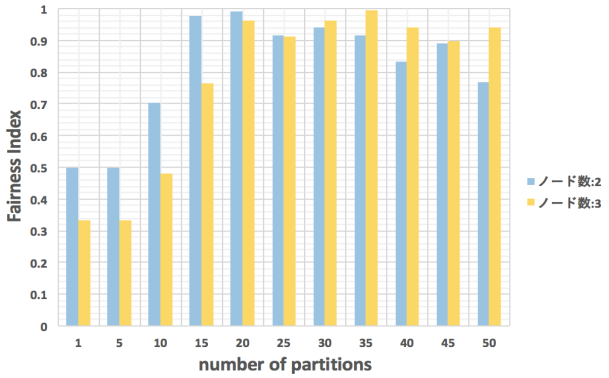

図 8 Fairness Index

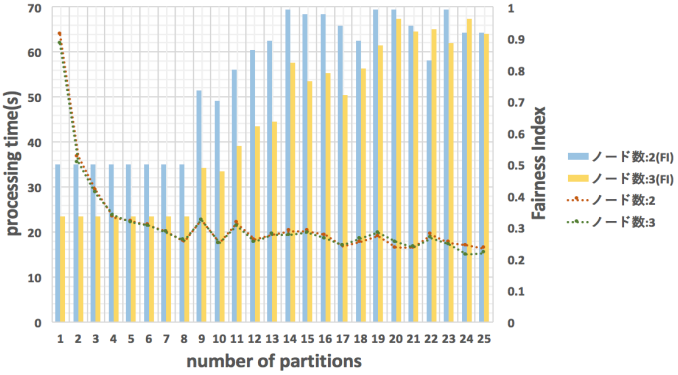

図 9 Fairness Index と実行時間変化

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 数が 5 から 15 にかけて公平性が大きく変化していることがわかる．

図 9 に，locality wait 時間 3 秒の状態で partition 数 25 まで細かく計測した際の Fairness Index 及び実行時間変化を示す．図 9 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 14 以上の場合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．

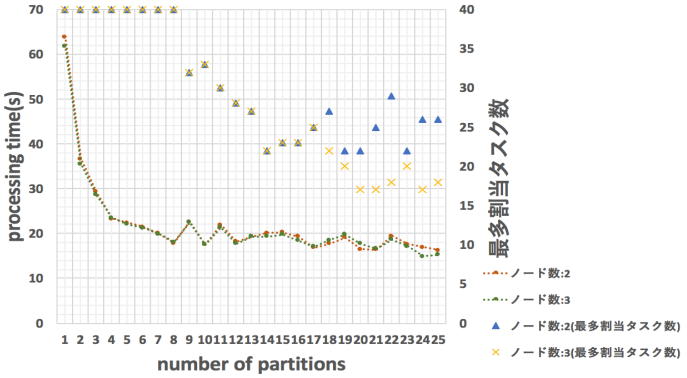

図 10 最多割り当てタスク数と実行時間変化

図 11 wait 0 秒の Fairness Index と実行時間変化

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 10 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 10 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 17 まで一致していた．また，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少ないと考えられる．

同様の計測を，locality wait 時間を 0 秒，1 秒と変化させて行った．以下に示す図 11 及び図 12 が wait を 0 秒に設定して測定した結果，図 13 と図 14 が wait を 1 秒に設定して測定した結果である．

結果から，locality wait 時間を 1 秒としたときにはデフォルトの 3 秒のときとほぼ同じようにタスクが分配されることがわかった．これに対して locality wait 時間を 0 秒としたときは，3 秒の時に生じていた partition 数 8 までのタスク分配の偏りが解消され，partition 数 2 以上でタスクが公平に複数ノードに分配されるようになることがわかった．

3.2.3 各タスクの処理時間

実行時間とタスク分配状況の計測結果から，実際に各タスクがどのノードに割り当てられ，処理にどの程度時間がかかっているのかを測定する実験を行った．図 15 にノード

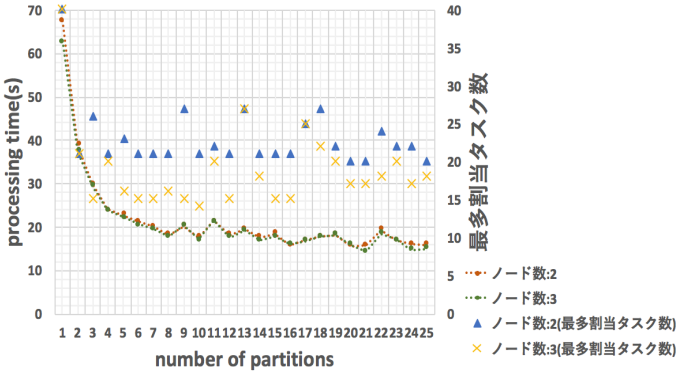

図 12　wait 0 秒の最多割り当てタスク数と実行時間変化

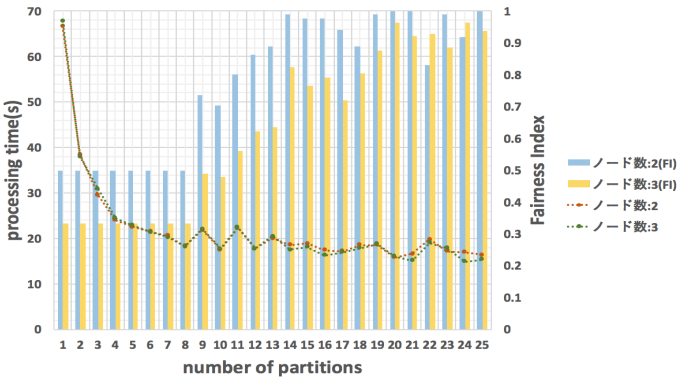

図 13　wait 1 秒の Fairness Index と実行時間変化

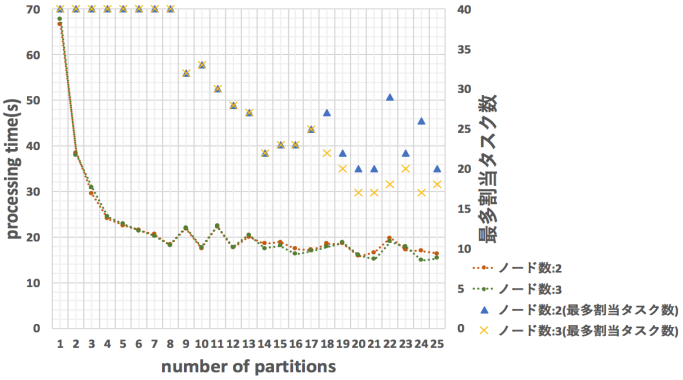

図 14　wait 1 秒の最多割り当てタスク数と実行時間変化

数 3，partition 数 9，locality wait 時間 wait3 秒の測定結果を，図 16 にノード数 3，partition 数 9，locality wait 時間 0 秒の測定結果を示す．どちらの場合もワーカノードを 3 つ接続しているが，wait を 3 秒に設定した場合は Locality を重視するのでノード 1 の 8 つのスレッド全てが埋まるまで他のノードにタスクが分配されることはなかった．図 15 では，ノード 1 の 8 つのスレッドにノード 2 のスレッド 1 つを加えた全 9 スレッドでタスクが処理されていることがわかった．これに対して locality wait 時間を 0 秒に設定した場合は Locality を考慮しないので 9 つの partition が 3 つのノードに均等に分配される．

図 17 にノード数 3，partition 数 40，wait3 秒の測定結果

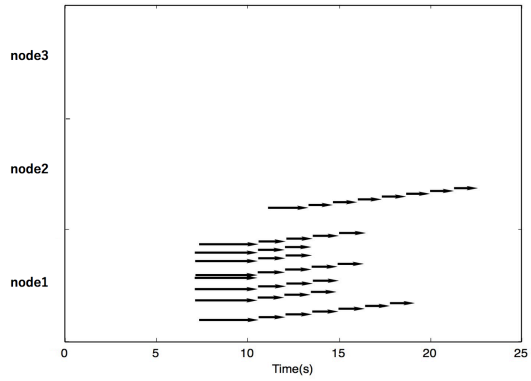

図 15　ノード数 3，partition 数 9，wait 3 秒のタスク処理時間

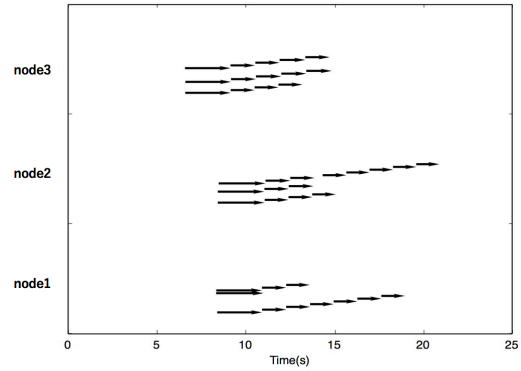

図 16　ノード数 3，partition 数 9，wait 0 秒のタスク処理時間

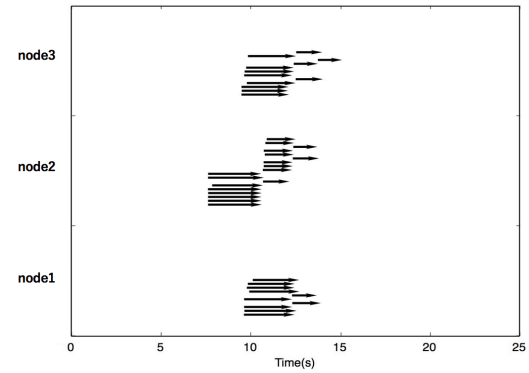

図 17　ノード数 3，partition 数 40，wait 3 秒のタスク処理時間

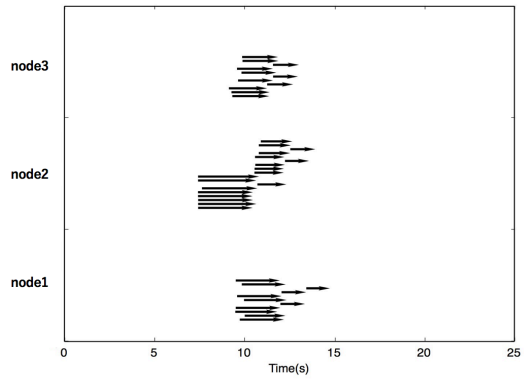

図 18　ノード数 3，partition 数 40，wait 0 秒のタスク処理時間

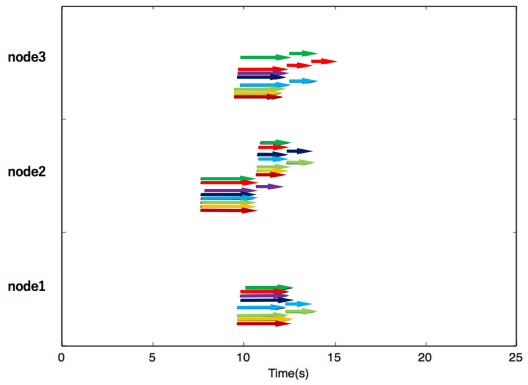

図 19 図 18 のタスク処理時間と見かけの partition 数

を，図 18 にノード数 3，partition 数 40，wait0 秒の測定結果を示す．図 17 と図 18 より，locality wait 時間が異なっていても partition 数が多くなるとほぼ同じ挙動になることが伺えた．前述の通り，partition 数の指定はプログラム中で partitionBy() というメソッドを用いることで指定できるが，見かけ上ジョブ全体は正確には指定 partition 数に分かれない．スレッド数やタスクの処理順序，処理時間の観点から図 17 のベクトルを partition ごとに色付けし，図 19 に示した．メソッド partitionBy() によって分割されたジョブは partition ごとにノードに割り当てられ，スレッド 1 つにつき partition1 つが処理される．割り当てられた partition 内の全てのタスクの処理を終えたスレッドは，新たな partition の処理を開始する．partition が均等に分割されればラウンドロビンに近い挙動になることが予測できるが，図 19 ではある 1 つの partition 内のタスクが 3 つに設定されてしまっているため，その partiton の処理に時間を要していることが伺えた．このような，タスクを多く内包する partition の処理時間による実行時間全体の律速が起きていることが，ノード数増加による実行時間の減少が見受けられない原因の 1 つと考えられる．

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，MapReduce[8][9] モデルに Map と Reduce 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．

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することで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．

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．実行時間とタスクの割り当て状況を調査し，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，locality wait 時間を変更することで Fairness と Locality にどの程度比重を置くかを考慮できることが判明した．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

本研究は一部，平成 29 年度国立情報学研究所公募型共同研究の助成及び，JSPS 科研費 JP16K00177，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) の委託業務及び JST CREST JPMJCR1503 の支援を受けたものです．

参考文献

[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 networks, 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).
[12] Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I.: Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In EuroSys 2010, (2010).
[13] ChainerMN, https://chainermn.readthedocs.io/en/latest/index.html.