Vol.2014-HPC-145 No.38
2014/7/30

# 分散プログラミング言語X10を用いたアナリティクスライブラリの実装と評価

千葉 立寛[1,a)]　竹内 幹雄[1,b)]　戸澤 晶彦[1,c)]

**概要**：大規模データを解析・分析するためのプラットフォームとしてHadoopをはじめとする様々な分散処理フレームワークが使われている．それに伴い，データマイニングなどで使われるアナリティクスアルゴリズム自身もスケーラブルに実行可能なように最適化する必要がある．現在利用されるライブラリの多くは，RやPython上に実装されているが，並列性を言語レベルで記述するように設計された言語ではないので，並列分散実行できるように書き換えることは困難である．また，ユーザーが記述可能な処理を限定するプログラミングモデルでは，簡単な処理を並列分散化するのには向いているが，新たに開発されたアルゴリズムを実装する際には，その制約により複雑なロジックを記述し難くなったり，それによって性能低下を引き起こす側面も持つ．一方，OpenMPやMPIなどを用いることで細かな最適化をアプリケーションユーザー自身が行うことが可能となるが，ハードウェアやシステム構成，データの送受信などに注意を払ってプログラミングする必要もあり，実装の生産性が低くなるという問題点もある．本稿では，アルゴリズムを実装する際の生産性と実行時の性能スケーラビリティを両立するための実行モデルとして，分散環境上でのアプリケーション実行モデルの1つであるPGASモデルに着目し，PGASおよび分散プログラミング言語X10でアナリティクスライブラリを記述するメリットについて論じる．それを踏まえ，X10を用いてHMMの学習アルゴリズムとDBSCANクラスタリングの並列分散化を行い，既存の実装に対する実行性能とスケーラビリティの比較を行った．

## 1. はじめに

本格的なビッグデータ時代の到来に対し，Hadoopに代表される大規模データ向け分散処理基盤の普及とともに，ビジネス，サイエンスの分野を問わず，様々な業界でビッグデータを活用した解析，分析がますます一般的になってきている．モバイルやセンサーデバイスから生成される画像やテキストなどの非構造化データ，人やモノの繋がりを示すグラフデータ，次世代シーケンサーから生成される大量のDNA塩基配列データなど，その種類 (Variety) は多岐に渡る．集約・蓄積したデータを解析・分析して新たな知見を効率よく迅速に導き出すためには，アナリティクスアルゴリズムそれ自身をスケールアウトによる性能向上が可能となるよう，並列分散環境での実行向けに最適化していくことが必要である．

大規模データを分析して異常値の判別や予測モデルを構築するために，データマイニングの様々な手法に多くの注目が集まっている．その中で使われる分類や回帰，クラスタリングなどデータ分析アルゴリズムの多くは，numpy，scipyなどの数値計算用ライブラリの充実や利用者が多いなどの理由から，RやPython上に数多く実装されており幅広く使われている．しかしながら，これらの言語は元々並列化を意識した言語ではなく，個々のアルゴリズムレベルでも並列分散実行可能なように作られてはいない．

MapReduceとHadoopフレームワークの登場により，アルゴリズムを並列分散実行させるためのハードルは以前よりも下がっており，文献 [1] で様々な機械学習アルゴリズムがMapReduceによって記述可能であることが示されたことにより，Mahoutなどの Hadoop向けのアナリティクスライブラリも開発されている．しかしながら，MapとReduceの2つのオペレーションのみで複雑な処理を書くことは可能ではあるものの，単純な処理に対してもMapReduceで記述しなければならず，その場合は多段のMapReduceで構成されるプログラムとなってしまう．実行ランタイムのオーバーヘッドを減らしたり，コンパイル時に段数の少ないMapReduceに変形する手法などがあるが，アルゴリズムによっては本質的に無駄の多い実行モデルになる場合もある．

[1] 日本アイ・ビー・エム（株）　東京基礎研究所
IBM Research - Tokyo
[a)] chiba@jp.ibm.com
[b)] mtake@jp.ibm.com
[c)] atozawa@jp.ibm.com

一方，マルチコア CPU や GPU などのアクセラレータ，高速なネットワークを備えた近年の並列分散システム上で，これらのアルゴリズムを実装する際には，ハードウェア性能を十分に発揮するためのプログラミングが必要になる．システム構成やアルゴリズムの特性を踏まえた上で，OpenMP や MPI，CUDA などのプログラミングモデルを適切に選択して並列分散実装を行うことになるが，データの送受信やメモリ上へのデータ配置などアルゴリズムの本質以外で考慮すべき点も多くなる．コードの保守性という観点ではコードの複雑性が増していくことは避けたいため，出来るだけプログラマビリティの高い見通しのよいプログラミングモデルでライブラリを実装する必要がある．

これらの背景を踏まえ，本稿では，アナリティクスライブラリ実装での生産性と実行性能スケーラビリティを両立するための実行モデルとして，分散環境上でのアプリケーション実行モデルの 1 つである PGAS (Partitioned Global Address Space) モデルに着目し，アナリティクスライブラリを PGAS 上で記述するメリットについて論じる．また，教師無し学習モデルの例として HMM[2] (Hidden Markov Model) を，クラスタリングの例として DBSCAN[3] (Density Based Spatial Clustering of Application with Noise) を選択し，PGAS モデルを採用する分散プログラミング言語 X10 を用いてこれら 2 つのアルゴリズムの並列分散化を行った．既存のアナリティクスライブラリと X10 実装での実行性能とスケーラビリティを IBM POWER7+マシンにて計測して比較を行ったところ，HMM は GHMM に比べて 3 倍，DBSCAN は scikit-learn に比べて 100 倍以上の高速化を実現した．また，入力データ数が 100 万のときの X10 単体でのスケーラビリティは，Place 数 1 での実行と比べて，HMM で 25 倍，DBSCAN で 30 倍までスケールすることを確認した．

## 2. 関連研究

機械学習や様々な統計解析を行うためのソフトウェアとして，数多くのアナリティクス手法や，アルゴリズムがモジュールとして実装されている R や Python などが幅広く使われている．例えば，Python においては，数値・行列演算や多次元配列などをサポートする numpy, scipy モジュール，および，クラスタリングや回帰などのアナリティクスライブラリを提供する scikit-learn モジュール [4] を適用することで，機械学習を簡単に実行することが可能である．さらに，インタラクティブに解析が行えるシェルが使えるため，解析対象データの特徴を抽出，予測モデルの構築や評価，新たな分析手法のプロトタイピングなどを行うのに適している．しかしながら，解析アルゴリズムは基本的にはシングルスレッドでの逐次実行となり，大規模なデータに対する実行ランタイムとしてはあまり適した環境ではない．R や Python においても，クラスタ環境で分散実行するための拡張モジュールも数多く提案されているが [5]，マスタースレーブ型で処理を分散させるなど比較的単純なものに限られ，複雑な処理を記述することは難しい．

並列分散環境上で大規模データに対する機械学習やグラフ解析を実行したいという要求は，近年非常に高まってきており，アルゴリズムの並列化や分散実行フレームワークに対する研究が活発である．Chu らの論文 [1] において，様々な機械学習アルゴリズムが，MapReduce[6] 上で並列実行可能であることが述べられており，このような背景から Hadoop 上で利用可能な機械学習ライブラリの Mahout[*1] が開発されている．しかしながら，Map と Reduce の 2 つのオペレーションのみで複雑な処理を書いた場合，比較的単純なオペレーションでも MapReduce の枠組みの中で記述しなければならず，プログラム全体が多段の MapReduce で構成されることもある．M3R[7] のようにメインメモリ上にデータを置いた MapReduce フレームワークなど Hadoop でのプロセス起動のオーバーヘッド減らしたり，コンパイル時に段数の少ない MapReduce に変形する手法などを用いて高速化することも提案されているが，アルゴリズムによっては本質的に無駄の多い実行モデルになる場合もあるため，より柔軟なオペレーションも実行可能であることが望ましい. [*2]

Spark[8] は，Scala で実装された分散処理フレームワークで，オペレータ (map, filter, count, reducebykey など) を連結させて処理を記述していくプログラミングモデルを提供する．フレームワークによって自動的にオペレータが並列化されるので，ユーザは入力データに対する処理ロジックにのみ集中して記述することが可能となり，プログラマビリティが高いシステムであると言える．繰り返し処理に使うデータセットを cache オペレータで明示的にインメモリに保存したり，RDD (Risillient Distributed Datasets)[9] と呼ばれるデータセットの欠損に対しての耐故障性を高める仕組みが実装されている．Hadoop/MapReduce よりもアナリティクス処理に適したシステムであり，Spark 上に実装された機械学習ライブラリ (MLlib) では，k-means や線形回帰などが利用可能である．しかしながら，Spark のランタイム上ではユーザレベルから非同期にタスクを実行する仕組みはサポートされていないため，タスク実行の非同期性を活かしたコードを記述しにくい．

GraphLab[10] は，C++で実装された分散処理フレームワークであり，計算とデータの関係をグラフで表現した graph-parallel モデルを採用している．グラフのノードに計算を，エッジにデータを当てはめ，ノードに繋がるエッジ上のデータを収集 (Gather) し，各ノードでの計算 (Apply) を行い，再びエッジにデータを戻す (Scatter) という処理を

[*1] https://mahout.apache.org/
[*2] 現在，Mahout は，MapReduce での開発を中止し，実行ランタイムを Hadoop から Spark にシフトしている．

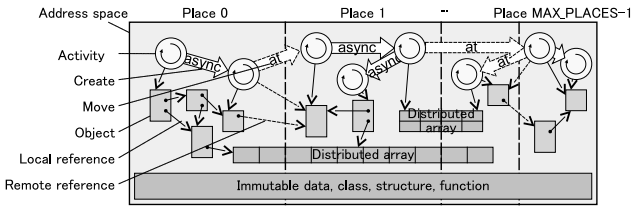


**図 1** X10 APGAS model

繰り返すことで計算を実行するモデルを採用する．各ノードでの演算は，ランタイムによって非同期に実行される．協調フィルタリングやクラスタリングといったアルゴリズムが GraphLab 上で実装されているが，その他のアルゴリズムについては，C++上で実装するか，GraphLab が提供する Java や Python の API を通じて graph-parallel モデルのプログラミングをする必要がある．

Spark のように複雑な処理をユーザーに記述させずに，分析や処理のロジックだけを書けるようにする DSL 指向のプログラミングモデルも数多く提案されている．Hadoop エコシステムにおける Pig や Hive などが代表例であり，SQL ライクな言語を用いるだけで，直接 MapReduce を書かなくても Hadoop 上で処理が実行される．大規模データ向けのデータマイニングに対しても，様々な DSL が定義されており，例えば，SystemML[11] では Matlab ライクな言語で，hivemall[12] では hiveSQL を用いて記述した回帰分析などの処理が Hadoop 上で実行される．また OptiML[13] は Scala 上に作られた機械学習向けの DSL であり，Scala や C++，CUDA のコードを生成することが出来る．単一ノード内での複数 CPU や CUDA へのオフロード機能を有しているものの，分散環境での実行は想定されていない．

## 3. X10：実行モデルと特徴

### 3.1 PGAS と X10 の実行モデル

X10[14] は，並列分散処理を言語単体で記述可能なプログラミング言語であり，並列分散環境を PGAS プログラミングモデルにより抽象化してアプリケーションユーザに見せることで，OpenMP や MPI, CUDA といった従来の HPC アプリケーションでのハードウェアを意識したプログラミングをせずに並列プログラムを記述可能にしている．また，並列分散プログラミングの生産性を高め，エクサスケールコンピューティングに向けた高い実行性能とスケーラビリティを両立させることを目標としている．[*3]

PGAS モデルにおいては，複数の計算ノードにまたがったグローバルなアドレス空間が提供され，各ノードにはそれぞれ分割したアドレス空間が配置されている．X10 においては，下記に示すように非同期実行のサポートを強化しているため，区別して APGAS (Asynchronous PGAS) モデルと呼んでいる．X10 における APGAS の概念を示したものが図 1 である．X10 においては，この分割したアドレス空間を *Place* と呼び，メモリの局所性が抽象化されている．データは異なる *Place* から透過的に参照可能であるが，実際に操作するためには同じ *Place* に移動してアクセスする必要がある．また，*Place* 内で非同期にタスク実行を行うための実行主体を *Activity* と呼び，軽量な Thread のように扱うことができる．X10 においては *async* 文により *Activity* を動的に生成したり，*at* 文を使うことで異なる *Place* に移動して *Activity* を実行する仕組みを有している．*Place* 内では，mutable なオブジェクトを生成し，複数の *Activity* で扱ってローカルでの共有メモリ型プログラムを記述できる一方，複数の *Place* 間で一意に存在して共有したいオブジェクトに関しては，異なる *Place* 内のオブジェクトを参照する *GlobalRef* と *atomic* 文の仕組みを使うことで，グローバルな共有データを用意することができる．

### 3.2 アナリティクスライブラリに向けて

2 節において，様々な機械学習向けのライブラリや分散処理フレームワークについて述べてきたが，(1) 分析・処理ロジックの記述しやすさ，(2) 分散処理フレームワーク上でのタスク実行の柔軟性，(3) 実行性能・スケーラビリティという 3 つの軸でのトレードオフが存在している．PGAS モデルおよびプログラムの生産性向上と高い実行性能の実現を目指す X10 は，これらの 3 つの要件を全て満たす言語であると考えている．表 1 に X10 および様々な並列分散処理フレームワークの特徴をまとめているが，本節ではこれらを踏まえ，アナリティクスライブラリを構築していく上で必要な要件と，X10 で実装することのアドバンテージについて説明する．

**データモデルと一貫性**

データの抽象化とそのデータの一貫性は，並列プログラムを書く上で非常に重要な要素の一つである．本稿では，データの共有モデルにのみ着目して論じる．Hadoop においては，Map や Reduce 処理をしている時にプロセス間でデータを共有する仕組みはフレームワークとしては用意しておらず，別途，Zookeeper[*4]などの外部の共有データストアを介する必要がある．Spark においては，RDD の仕組みとして Read-Only なデータであり，データに対する *Transformation* という形で実行時に関数が同期的に実行されて評価される．cache() を呼び出すことで，再利用したいデータをメモリ上に保存するオペレーションは用意されているが，ユーザ側で明示的に他のプロセス間で共有したいデータをメモリ上に保存するためのオペレーションは操作は存在していない．GraphLab では，グラフ上のノードとエッジにデータが存在し，依存関係にあるノード同士

[*3] http://x10-lang.org/

[*4] http://zookeeper.apache.org/

表 1　並列分散処理言語・フレームワークの比較

| Framework | Programming Model | Programming Lang. | Communication | interop | Asynchronous | Resiliency |
|---|---|---|---|---|---|---|
| X10 | APGAS | X10 | MPI,PAMI,Socket | Java,C++,CUDA | yes | yes |
| Hadoop | MapReduce | Java | IPC/RPC | - | no | yes |
| Spark | Scala DSL/RDD | Scala | Socket | Java,Python | no | yes |
| GraphLab | graph-parallel | C++ | Socket,MPI | Java,Python | yes | no |

でデータが共有される．共有されるデータの属性として，ユーザーは 3 段階（範囲）の一貫性レベルを設定し，そのルールに従ってランタイムでは実行スケジューリングが行われる．X10 においては，ユーザーが明示的に GlobalRef と atomic 文を用いることで，データの共有と一貫性を確保している．

**並列プログラミングモデル**

表 1 に示すフレームワークにおけるプログラミングモデルは，処理をどのようにユーザーに抽象化して見せるか，という点でそれぞれ特色を持っている．この抽象化は，そのままユーザーが記述可能な処理の制約条件となってくるため，プログラムを記述する柔軟性と生産性をすり合わせることが必要になってくる．機械学習などで使われる計算の多くが，与えられたデータセットに対してパラメータが収束するまで定義した関数を繰り返し適用させるような処理が基本となる．そのため，仮に Map，ReduceByKey，Join，Filter などの決められたオペレーションのみがサポートされたフレームワークであっても，十分にアルゴリズムを記述可能であるが，細かな制御をユーザー側で行うことを犠牲にしている．Spark や GraphLab に関しては，実行ランタイム側 (RDD, Task Scheduler) が並列性を吸収しているので，ユーザーからはあまり並列実行に対して意識する必要はない．一方 X10 は，あくまでフレームワークではなく言語なので，並列化のために必要なコード量は少ないものの，ユーザーが並列に実行したい部分を明示的に指示することが必要である．

**他言語や既存ライブラリの利用**

X10 で記述することの優位性として，実行ランタイムのインターオペラビリティの高さが挙げられる．X10 では，C++にコンパイルされてネイティブ実行される Native X10 と，Java にコンパイルされて JVM 上で実行される Managed X10[15] の 2 つのモードがあり，ユーザーの実行環境や既存のライブラリの利用可否によって柔軟にランタイムを選択可能である．X10 の型システムは Java の型システムを含んでおり，Java の型は X10 へ直接 import 文を追加することが可能であるため，Managed X10 では既存の Java のライブラリをシームレスに扱うことができる．Native X10 では X10 のコードを CUDA にコンパイルし，GPGPU 上で実行可能とする機能を備えている [16]．ネイティブ向けのライブラリの利用に関しても*@Native* ディレクティブを付加するだけで呼び出すことが可能である．X10 で書かれたライブラリも様々なものが利用可能となってきており，グラフ計算向けの ScaleGraph[17] や，内部で BLAS を呼び出して演算を行う行列計算向けの GML などがある．

**耐故障性**

様々なレベルでの耐故障性を確保することは，アナリティクスアルゴリズムを並列分散実行する際においても重要である．フレームワークによって，様々なレベルでの耐故障性をサポートしているが，X10 においてはノードが故障してそのノードが担当していた Place に係るアクティビティやデータが消失した場合でも，残存する Place のみで処理を継続することが可能にするための言語拡張である Resilient X10[18] が開発されている．処理途中でデータの一部や計算結果が不足していても継続可能なアルゴリズムも多く，最終的に得られる予測モデルや値の精度の低下が許容できる範囲内であれば，実行を最後まで継続させたいという要求は多い．Resilient X10 において，発生した障害 (DeadPlaceException) を受け取ったときに残りの Place 間適切に処理することで期待される耐故障性を X10 においても確保可能である [19].

## 4. Case Study: Example Analytics Application

SVM や決定木，回帰分析，クラスタリングなど，一般的に使われるデータタマイニングアルゴリズムは非常に多岐に渡っており，本稿で様々なアルゴリズムを網羅することはできない．大規模データマイニングの応用アプリケーションの一つとして異常検知があるが，大量のデータの中から正常でないと判断されうるデータを検出するために，クラスタリングを用いる手法や，データが時系列データの場合には予測モデルを生成して，観測されたデータに対するスコア（尤度）を元に判断する手法などが考えられる．本稿では，クラスタリングの手法として DBSCAN を，時系列データに対するモデルとして HMM を考え，これら 2 つのアルゴリズムを例題に，X10 上でのアナリティクス手法の実装と並列化に関する説明を行う．

### 4.1 Hidden Markov Model

**アルゴリズムと並列化**

Hidden Markov Model (以下，HMM) は，時系列データを解析する際に用いられる統計モデルの一つであり，音

声認識やジェスチャーの認識などの分野で幅広く応用されてきた．また，近年では，ネットワークの侵入検知やイベントデータの誤り検知など異常検知への応用や，たんぱく質のモチーフ検索に HMM を用いるなど，バイオインフォマティクス分野においても幅広く使われている．

本稿では，HMM のパラメータを文献 [2] にならって以下のように定義する．HMM の内部パラメータは，観測される状態の数を O，隠れ状態の数を S としたとき，初期状態確率ベクトル $\pi = (\pi_1, \pi_2, \cdots \pi_s)$，状態 $S_i$ から $S_j$ に遷移する遷移確率 a(i, j) (i ∈ S, j ∈ S)，各状態での観測データを生成する出力確率 b(k, l) (k ∈ S, l ∈ O) の 3 つのパラメータが存在する．パラメータ未知の HMM の学習フェーズでは，観測データ（シンボル列）を入力として，尤度が最も高くなるようなパラメータを推定するが，このとき使われる代表的なアルゴリズムとして，Baum-Welch アルゴリズムが良く知られている．入力シンボル列 Obs に対して，Forward, Backward アルゴリズムを用いて尤度を計算し，時刻 t における状態 i から状態 j への遷移確率 $\Gamma_t(i, j)$ を計算する期待値フェーズと，$\Gamma_t(i, j)$ を用いてパラメータ a,b を再計算して尤度が最大となるように更新する最大化フェーズで構成される．

入力シンボル列が複数存在する場合，各入力列に対して同様の計算を行い，パラメータ a, b に足しこんでいき，全てのシンボル列に対する計算が終わった段階で正規化する．その後，生成されたモデル $M_i$ を用いて尤度を計算し，前回のモデル $M_{i-1}$ との差が収束条件を満たさない場合，再びパラメータの学習を行う．個々の入力シンボル列の計算に依存関係がなく，イテレーションの終了判定を行うタイミングでパラメータの更新と共有を行えばよい．学習アルゴリズムの並列化の概要を Algorithm1 に示す．並列度を P，シンボル列を N としたとき，(step.1) 入力シンボル列 $N$ を均等 (N/P) に分割してローカルプロセスにロードし，(step.2) 初期モデル $M_0$ をセットした後，全プロセスにコピーする．(step.3) 与えられた入力シンボル列に対してローカルのモデルパラメータ $M_{local}$ を更新し，(step.4) 他のプロセスでの計算されたパラメータをリダクション処理してモデル $M_{new}$ を生成する．(step.5) 前のモデル $M_{target}$ とのパラメータ誤差 $eps$ を計算し，終了条件を満たさない場合，(step.3) に戻り計算を再開する．

**X10 での実装**

Algorithm 1 でも示した通り，ローカルでの Baum-Welch 実行中にデータ交換をする必要はなく，初期モデルを $M_0$ をブロードキャストをする時と，モデル更新のタイミングで個々の $M_{local}$ のデータをリダクション処理して $M_{new}$ を生成時の AllReduce 通信の 2 回のみ通信が発生する．Algorithm 1 を X10 で記述した場合のコード例を図 2 に示す．Trainer を全 Place に生成し (line 24)，at async にて非同期の SPMD タイプの処理により，学習フェーズをスタートさせる．(line 28)．初期モデルブロードキャスト部分 (line 38 - 41) では，Place 0 で生成した $M_0$ を各 Place の Trainer インスタンスの startModel にセットする操作が記述されているが，send/recv 型のデータ転送処理を記述する必要はなく，オブジェクトへの代入を行うような操作で各 Place のメモリにコピーされる．このように，データの転送や処理の分散化に対して直感的に記述できるため，結果的に，設計したアルゴリズムと記述するコードの乖離が非常に少なくて済むように分散処理を記述可能である．

**Algorithm 1** Parallel Baum-Welch Algorithm

```
 1: procedure TRAIN(a, b, π, N)
 2:     data ← Load(N)                          ▷ step.1: load equally divided chunk
 3:     M_0 ← Init_Model(a, b, π)
 4:     M_target ← Broadcast(M_0)               ▷ step.2: copy M_0 to all
 5:     for i ← 0; i < iterations; i ← i + 1 do
 6:         M_local ← BaumWelch(M_target, data) ▷ step.3: start training
 7:         M_new ← AllReduce(M_local)          ▷ step.4: produce M_new
 8:         eps ← Distance(M_target, M_new)     ▷ step.5: calc. eps
 9:         if eps ≤ ε then
10:             return M_new
11:         else
12:             M_target ← Broadcast(M_new)     ▷ update M_target
13:         end if
14:     end for
15: end procedure
```

### 4.2 DBSCAN

**アルゴリズムと並列化**

DBSCAN[3] は，k-means などと並ぶクラスタリングアルゴリズムの一手法で，クラスタリング対象の点集合間の接続関係（距離）に基づいて分類していくことで，k-means では表現できない任意形状のクラスタを抽出することが可能となるアルゴリズムである．DBSCAN 実行時にユーザーが入力するパラメータは，$minpts$ と $eps$ の 2 つである．$minpts$ はクラスタを構成するために必要な最小の点の数を表し，$eps$ は 2 点間の距離 $dist(x, y)$ に対して，$dist(x, y) \leq eps$ の条件を満たすような点 y をクラスタの構成要素とするための閾値である．オリジナルの DBSCAN では，点集合 $X$ の任意の点 $x \in X$ から開始して，点 $x$ の近傍集合 ($N = Neighbors(x, eps)$) とその集合内の点 $x'$ に対する近傍集合 N′ を探索しながら点 $x$ のクラスタに属する集合の領域を増やしていくシーケンシャルなアルゴリズムになっている．

Patwary らが DBSCAN の並列化手法を文献 [20] にて提案しており我々もこの手法をベース DBSCAN を並列化した．全てのデータポイントには，ユニークな ID が存在すると仮定し，また，全てのデータポイントがそれ自身のみを含む *Disjoint-set* な *Singleton Tree* を構成しているとする．点 x の近傍集合 ($N = Neighbors(x, eps)$) を探索して，$minpts \leq |N|$ であるときに，$y \in N$ がまだどのクラスタにも

```
class Model(S:Int,O:Int){
  val a = new Matrix(S,S);
  val b = new Matrix(S,O);
  val pi = new Vector(S);
  ... //その他メソッド
}
class Driver {
  public static def main() {
    //観測状態数 3,隠れ状態数 3 の HMM の学習
    val myModel = Trainer.run(3,3,fname);
  }
}
class Trainer(S:Int,O:Int) {
  public static type State = PlaceLocalHandle[Trainer];
  //観測データ列を Obs として定義
  public static type Obs = Rail[Int];
  //trainer が処理する観測データ列を保存する配列
  public var data:Rail[Obs] = null;
  public var startModel:Model = null;
  private val epsilon = 0.01;

  public static def run(S:Int,O:Int,fname:String):State {
     val everyone = Place.places();
     val roots = PlaceLocalHandle.make[Trainer](everyone,
     ()=>new Trainer(S,O));
   finish { for (p in everyone) { at(p) async {
     val runner_ = roots();
       runner_.loadData(fname); //step.1
       runner_.train(roots);    //各 Place で学習フェーズを開始
     }}}
        return roots;
  }
  public def loadData(fname:String):void {
    // N/P のサイズに区切ったデータをロード
  }
  public def train(roots:State):void (
    if(here.id == 0) {
      //step.2 初期モデルのコピー
      val startModel:Model = makeModel(S,O);
      finish { for (q in Place.places()) { at(q) async {
        roots().startModel = startModel;
      }}}
    }
    this.tmpModel = this.startModel;
    for (var i:Long = 1; i < maxIters; i++) {
       val beforeModel = this.tmpModel;
       // step.3 学習フェーズ
       val localModel = this.startModel.baumWelch(data);
       Clock.advanceAll(); // wait all
       // step.4 パラメータを合成したモデルの生成
       val newModel = AllReduce(localModel);
       // step.5 終了条件チェック
       val eps = Diff(beforeModel,newModel);
       if (eps < epsilon) return;
     else this.tmpModel = newModel;
    }
  }
}
```

**図 2** X10 で Algorithm1 を実装したときの擬似コード

属していなければ，y と x の tree を結合 (Union 操作) することで，クラスタに属するデータポイントを増やしていく．*Disjoint-set* に対する Find 操作は，クラスタを構成するツリーのルートポイント探索する操作を意味し，Union 時には，2 つのクラスタのルートポイントの ID を比較し，その大小関係からルートポイントの付け替えを行うことで，新たなクラスタを構成するツリーとする．並列実行の方針としては，データポイントをプロセスに対して分割し，各プロセス内のローカルデータに対してクラスタリング（Local Union）を行う．その後，各プロセスが担当している領域から eps 離れた領域 (外側) に存在しているポイントに対して，結合可能かの探索を行い，条件を満たすポイントに対してクラスタリング (Remote Union) を行う．

**X10 での実装**

基本的には，データを各 Place に分割して SPMD 型で並列計算を実行 (Local Union フェーズ) し，その後で各 Place 間での Remote Union フェーズを記述することになる．X10 での実装のポイントとしては，データポイントの Place 間での分布 (Dist) を定義することである．例えば，Remote Union フェーズでは各 Place が担当する領域の eps 分だけ外側に存在するポイントに対して結合操作をしていくが，これらのポイントは別 Place で管理されている．MPI で実行する場合，個別にランクを指定して境界領域に存在するポイントに対する操作を send/recv しなければいけないため，非常に煩雑なコードになる．ポイントとそのポイントが存在する Place の関係が定義されている X10 では，ポイントが存在する Place へ at 文を用いて移動して結合操作を実行するというロジックで記述できるため，煩雑なデータ交換ロジックを記述することなく，シンプルで見通しの良いコードが記述できる．

## 5. 性能評価

X10 を用いて実装した HMM と DBSCAN の 2 つのアルゴリズムの性能について，既存のアナリティクスライブラリとの比較を行う．HMM を比較するためのアナリティクスライブラリは C 言語で実装された GHMM[*5]，および Python で実装された scikit-learn[*6] を用い，DBSCAN に対しては Java で実装された Commons.Math ライブラリ[*7]と，HMM と同様に scikit-learn を用いた．

実験環境には IBM Flex System p260 (7895-23X) を 3 ノード用いた．各ノード間は 10Gb Ethernet で接続されている．各ノードは 4-Way SMT をサポートした 4.1GHz の IBM POWER7+ プロセッサ (8 コア) を 2 ソケット (合計 16 コア) 搭載し，メモリが 128GB，OS は RedHat Enterprise Linux 6.5 が動いている．X10 のバージョンは 2.4.3.1 を使用した．X10 は Native-backend, Managed-backend の両バックエンドを用意した．Native へのコンパイルには IBM XLC V12.1 コンパイラを使用し，コンパイルオプションは `"-O -NO_CHECKS -x10rt sockets"`を設定した．また，Managed では IBM Java 1.7.0 (SR7) を使用した．計測結果は 10 回計測したときの最小値を採用している．

[*5] http://ghmm.org/
[*6] http://scikit-learn.org/
[*7] http://commons.apache.org/proper/commons-math/

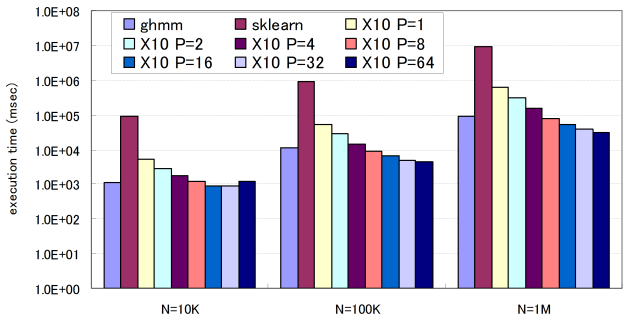


**図 3** GHMM, scikit-learn, X10 での HMM の実行性能比較 (1 ノード)

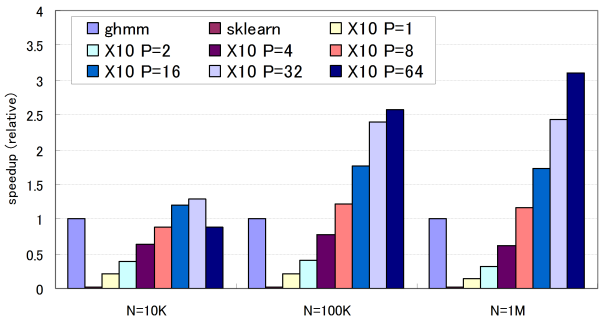


**図 4** GHMM に対する scikit-learn, X10 のスケーラビリティ性能 (1 ノード)

### 5.1 HMM

図 3 は，HMM への入力サンプル列のサイズを変化させたときの GHMM, scikit-learn および X10(Native) での Baum-Welch アルゴリズムを用いた学習フェーズの実行性能を比較している．HMM のパラメータは，状態数と観測数を共に 10 と設定し，入力サンプルの数は 10K, 100K, 1M の 3 つのデータセットを用意し，それぞれ観測列の長さは 10 である．X10 は 1 ノードのみを用い，Place 数 1 のときが逐次実行で，Place 数 2 から 64 までは並列化したときの性能を表している．

逐次実行においては GHMM が最も高速であり，scikit-learn に対しては 100 倍，X10 に対しても 5 倍ほど高速であった．この性能差は実装上の違いに起因するもので，scikit-learn に関しては，C 拡張と BLAS ライブラリを呼び出す機能が用意されているが，hmm 実装においては使われておらず python 上でのみの実行となったためである．X10 においても，2 次元配列に対して直接計算を繰り返す実装となっており，行列計算向けのライブラリ (GML) を使った高速化や，各 Place へのデータ分割後の計算を async 文を用いた非同期実行させるなど，HMM のパラメータの計算に関しての最適化を行っていない．

次に X10 で並列化した Baum-Welch アルゴリズムの性能を比較する．図 3 では並列化した際の実行時間を表し，図 4 では GHMM の性能を 1 としたときの性能向上を表している．GHMM に比べて，X10 の Place 数を 8 としたときに同程度の実行性能となり，以後，Place 数を増加させ

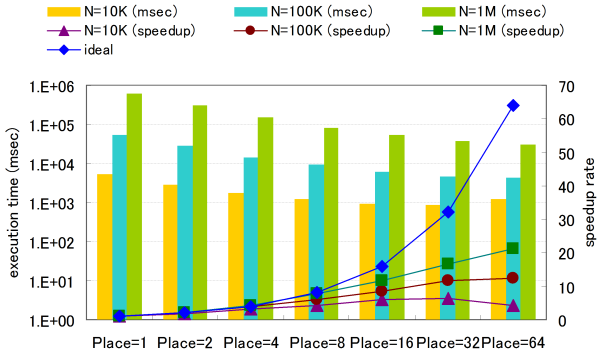


**図 5** X10 HMM のスケーラビリティ性能 (1 ノード)

るに従い性能が向上していった．比較的小さな入力データ数では計算量があまり多くないため，並列化した場合でも高々 1.25 倍の性能向上であるが，データ数が多くなるにつれて並列性能が向上し，N=1M のときに 3 倍以上高速化された．また，図 5 では，1 ノード上での X10 実装の Place 数に対するスケーラビリティを評価している．Place あたりのデータ量が小さくなりすぎない範囲であれば非常によくスケールし，N=1M の場合，Place 数 1 に対して Place 数 64 では 20 倍まで性能がスケールすること確認した．

### 5.2 DBSCAN

同様に DBSCAN に関してもポイント数を変えたデータセットを 3 つ用意し，それぞれに対して commons-math, scikit-learn および X10(Managed) での DBSCAN アルゴリズムの実行性能を比較した．X10 は Patwary らが提案している DBSCAN を実装しているのに対し，他の実装は DBSCAN のオリジナルアルゴリズムを実装しているため，内部で動作するアルゴリズムそのものが異なっている．DBSCAN のパラメータは minPts=2，eps=0.01 に設定し，データポイントは同一の点にならないよう調整された 2 次元のデータを用いた．

図 6 ではそれぞれの実装での 1 ノードを用いたときの実行時間を比較している．scikit-learn などと比べて，データ数の大小に関わらず常に X10 実装のほうが高速に実行できることを確認した．scikit-learn の性能を 1 とした場合，Place 数 1 であっても，N=10K で 2 倍，N=1M で 5 倍程度の高速化を実現しているが，これは，ベースになっているアルゴリズムが異なっているためである．次に並列化したときの性能を比べると，N=1M のときに 100 倍以上の高速化を実現している．これは，我々の DBSCAN の実装において，Place 間で非同期にデータ交換したり，Place 内で多数の Activity を生成して非同期にクラスタを構築したりするため，Place の数以上の並列性が存在しているためである．それゆえ，1 ノード (64 コア) 上での実験では，Place 数 16 程度のときが性能が最も良く，以降オーバーヘッドにより性能が低下することを確認した．

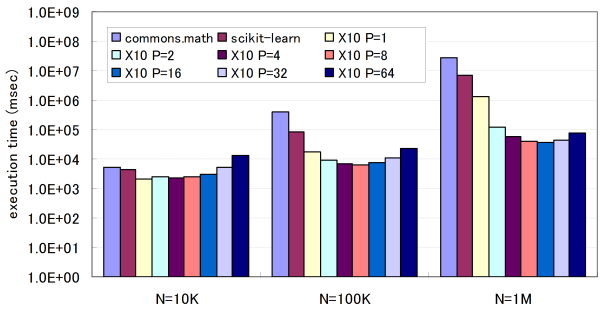


**図 6** commons.math, scikit-learn, X10 での DBSCAN の実行性能比較 (1 ノード)

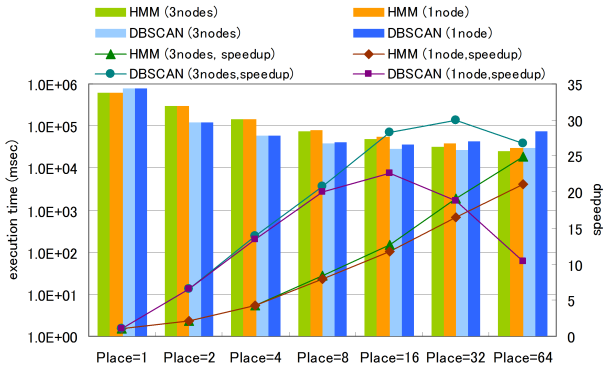


**図 7** 1 ノードおよび 3 ノード使ったときの HMM と DBSCAN の Place 数に対する性能スケーラビリティ

最後に，HMM および DBSCAN のスケーラビリティ性能を確認するため，1 ノードと 3 ノードでの実行性能を比較した．この結果を図 7 に示す．HMM では各 Place では Activity を生成せず，ほぼ各 Place での計算が独立に実行されるため，1 ノードと 3 ノードでの性能差があまりないが，Place 数 16 以降では 3 ノードを用いたときのほうが性能が向上し，Place 数 1 のときと比較して Place 数 64 では約 25 倍の性能向上が確認された．DBSCAN においては，1 ノードのときは Place 数 16 以降性能が低下していったのに対し，3 ノード用いたときは Place 数 32 まで性能が向上した．Place 数 32 のときの性能は，Place 数 1 のときと比較して約 30 倍の性能向上を確認した．

## 6. おわりに

本稿では，アルゴリズム実装での生産性と性能スケーラビリティを両立するための実行モデルとして，分散環境上でのアプリケーション実行モデルの 1 つである PGAS モデルに着目し，アナリティクスアルゴリズムを PGAS 上で記述するメリットについて論じた．また，分散プログラミング言語 X10 を用いて HMM の学習アルゴリズムと DBSCAN クラスタリングの並列分散化を行い，既存の実装に対して実行性能とスケーラビリティの比較を行った．Place 数を増やしていったとき，HMM は GHMM に比べて 3 倍，DBSCAN は scikit-learn に比べて 100 倍以上の高速化を実現した．また，それぞれのアルゴリズムにおける Place 数 1 のときの性能に対しては，HMM で 25 倍 (Place 数 64)，DBSCAN で 30 倍 (Place 数 32) までスケールすることを確認した．

今回，環境に依存しないような実装を優先したため，GML や CUDA の機能を使う実装は行っていないが，今後は，HMM や DBSCAN の実装で GML や CUDA を使った高速化も検討する予定である．また，Spark などの他の分散実行フレームワーク上に実装した並列実行可能な HMM や DBSCAN との性能比較や，回帰や SVM といった他のアルゴリズム，および，より大規模かつ実データを使ったアプリケーションシナリオに対する評価実験を行う予定である．

## 参考文献

[1] Chu, C. T., Kim, S. K., Lin, Y. A., Yu, Y., Bradski, G. R., Ng, A. Y. and Olukotun, K.: Map-Reduce for Machine Learning on Multicore, *NIPS* (Schölkopf, B., Platt, J. C. and Hoffman, T., eds.), MIT Press, pp. 281–288 (2006).

[2] Rabiner, L.: A tutorial on hidden Markov models and selected applications in speech recognition, *Proceedings of the IEEE*, Vol. 77, No. 2, pp. 257–286 (1989).

[3] Ester, M., Kriegel, H.-P., Sander, J. and Xu, X.: A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise., *KDD*, AAAI Press, pp. 226–231 (1996).

[4] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. and Duchesnay, E.: Scikit-learn: Machine Learning in Python, *Journal of Machine Learning Research*, Vol. 12, pp. 2825–2830 (2011).

[5] Venkataraman, S., Bodzsar, E., Roy, I., AuYoung, A. and Schreiber, R. S.: Presto: Distributed Machine Learning and Graph Processing with Sparse Matrices, *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, ACM, pp. 197–210 (2013).

[6] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, Berkeley, CA, USA, USENIX Association, pp. 10–10 (2004).

[7] Shinnar, A., Cunningham, D., Saraswat, V. and Herta, B.: M3R: Increased Performance for In-memory Hadoop Jobs, *Proc. VLDB Endow.*, Vol. 5, No. 12, pp. 1736–1747 (2012).

[8] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S. and Stoica, I.: Spark: Cluster Computing with Working Sets, *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, USENIX Association, pp. 10–10 (2010).

[9] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S. and Stoica, I.: Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing, *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, USENIX Association, pp. 2–2 (2012).

[10] Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A. and Hellerstein, J. M.: Distributed GraphLab: A

Framework for Machine Learning and Data Mining in the Cloud, *Proc. VLDB Endow.*, Vol. 5, No. 8, pp. 716–727 (2012).

[11] Ghoting, A., Krishnamurthy, R., Pednault, E., Reinwald, B., Sindhwani, V., Tatikonda, S., Tian, Y. and Vaithyanathan, S.: SystemML: Declarative Machine Learning on MapReduce, *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, IEEE Computer Society, pp. 231–242 (2011).

[12] Yui, M. and Kojima, I.: A Database-Hadoop Hybrid Approach to Scalable Machine Learning, *Big Data (BigData Congress), 2013 IEEE International Congress on*, pp. 1–8 (2013).

[13] Sujeeth, A., Lee, H., Brown, K., Rompf, T., Chafi, H., Wu, M., Atreya, A., Odersky, M. and Olukotun, K.: OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning, *Proceedings of the 28th International Conference on Machine Learning (ICML-11)* (Getoor, L. and Scheffer, T., eds.), ICML '11, ACM, pp. 609–616 (2011).

[14] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C. and Sarkar, V.: X10: An Object-oriented Approach to Non-uniform Cluster Computing, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, ACM, pp. 519–538 (2005).

[15] Takeuchi, M., Makino, Y., Kawachiya, K., Horii, H., Suzumura, T., Suganuma, T. and Onodera, T.: Compiling X10 to Java, *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, ACM, pp. 3:1–3:10 (2011).

[16] Cunningham, D., Bordawekar, R. and Saraswat, V.: GPU Programming in a High Level Language: Compiling X10 to CUDA, *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, ACM, pp. 8:1–8:10 (2011).

[17] Dayarathna, M., Houngkaew, C., Ogata, H. and Suzumura, T.: Scalable performance of ScaleGraph for large scale graph analysis, *HiPC*, IEEE, pp. 1–9 (2012).

[18] Cunningham, D., Grove, D., Herta, B., Iyengar, A., Kawachiya, K., Murata, H., Saraswat, V., Takeuchi, M. and Tardieu, O.: Resilient X10: Efficient Failure-aware Programming, *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, ACM, pp. 67–80 (2014).

[19] Kawachiya, K.: Writing Fault-Tolerant Applications Using Resilient X10, *Research Report (RT0960)*, IBM Research - Tokyo (2014).

[20] Patwary, M. A., Palsetia, D., Agrawal, A., Liao, W.-k., Manne, F. and Choudhary, A.: A New Scalable Parallel DBSCAN Algorithm Using the Disjoint-set Data Structure, *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Los Alamitos, CA, USA, IEEE Computer Society Press, pp. 62:1–62:11 (2012).