Vol.2020-HPC-175 No.5
2020/7/30

ベクトルプロセッサを用いた決定木学習の高速化

大道 修[1,a)]　荒木 拓也[1,b)]

概要：本稿では，ベクトルプロセッサに適した処理方式により決定木学習を行う手法を提案し，SX-Aurora TSUBASA を用いて決定木学習を高速化した結果について報告する．決定木学習は学習データが所定の条件を満たしているか否かの判断を要する演算が大半を占めており，提案手法では多数の条件分岐が関わる演算を行列積演算で代替することで処理を高速化する．性能評価の結果，x86 CPU 向け並列分散処理ミドルウェアである Apache Spark を用いた決定木学習に比べ，SX-Aurora TSUBASA を用いた決定木学習は最大 20 倍高速であることを確認した．

Acceleration of Decision Tree Learning on Vector Processor

Abstract: In this report, we propose a method to accelerate decision tree learning for vector processors and show results of acceleration with SX-Aurora TSUBASA. The most part of decision tree learning requires to check whether learning data satisfy several conditional expressions. In order to utilize vector processors effectively, our method replaces many conditional branches by matrix multiplication. We compare our method with SX-Aurora TSUBASA against Apache Spark MLlib, which is a parallel and distributed machine learning library for x86 CPUs. As a result, we confirm that our method can perform decision tree learning up to 20 times faster than MLlib.

1. はじめに

決定木（Decision Tree）は，非線形回帰やクラス分類，ランキング学習などに有効な機械学習モデルのひとつである．学習結果が木構造グラフで表されるモデルのため，推論の過程を人間が目で見て理解しやすく，予測結果が求められるに至った理由を解釈することが重要な場合や，詳細な学習モデル設計の前に学習データの大まかな傾向を捉えたい場合などによく利用されている．また，複数の決定木を束ねたアンサンブル学習であるランダムフォレスト（Random Forest）[1; Chapter 15] や GBDT（Gradient Boosting Decision Tree） [1; Chapter 10] は，場合によっては深層学習に匹敵する精度が得られることもある [2]．近年は学習データの大規模化が進んでおり，決定木アンサンブル学習においても，数ギガバイト規模のデータに対して一秒未満〜十数秒程度で決定木一本分の学習（1 イテレーション）を実行できることが求められている [3, 4]．

大規模データを高速に処理するには SIMD（Single Instruction Multiple Data）型プロセッサの活用が有効であり，機械学習においてもベクトルプロセッサや GPU などの SIMD 型プロセッサを活用した高速化が広まっている．SIMD 型プロセッサでは，ひとつの命令を複数のデータ要素に対して一斉に実行することで効率的に処理を行うことができる．このような一斉処理は行列演算と特に相性が良く，科学技術計算には古くからベクトルプロセッサが用いられてきた．また，深層学習も中核となる処理が膨大な行列演算に帰着され，GPU を活用した深層学習の高速化が広く行われている．しかし決定木の学習においては，学習データが所定の条件を満たしているか否かの判断を要し条件分岐を行うことが前提の演算が大半を占めており，そのままでは SIMD 型プロセッサの能力を有効に活用することが難しいという問題がある．SIMD 型プロセッサの能力を最大限に活用するためには，一斉処理が困難な部分をできるだけ減らし，データ処理プログラム中の大部分を SIMD 方式の演算に置き換えることができる工夫が必要である．

本稿では，NEC のベクトルプロセッサを搭載した SX-Aurora TSUBASA[*1] を用いて決定木学習を高速に実行することができる手法を提案する．以下ではまず 2 章で決定

[1] 日本電気株式会社
NEC Corporation
[a)] o-daido@nec.com
[b)] takuya_araki@nec.com

[*1] https://jpn.nec.com/hpc/sxauroratsubasa/

© 2020 Information Processing Society of Japan

表 1　学習データの例 [5]
Table 1　An example of a dataset

Feature				Label
#1	#2	#3	#4	
Slashdot	US	Yes	18	None
Google	FR	Yes	23	Premium
Slashdot	UK	No	21	None
Digg	US	Yes	24	Basic
Kiwitobes	FR	Yes	23	Basic
Google	UK	No	21	Premium
(N/A)	NZ	No	12	None
(N/A)	UK	No	21	Basic
Google	US	No	24	Premium
Slashdot	FR	Yes	19	None
Digg	US	No	18	None
Google	UK	No	18	None
Kiwitobes	UK	No	19	None
Digg	NZ	Yes	12	Basic
Google	UK	Yes	18	Basic
Kiwitobes	FR	Yes	19	Basic

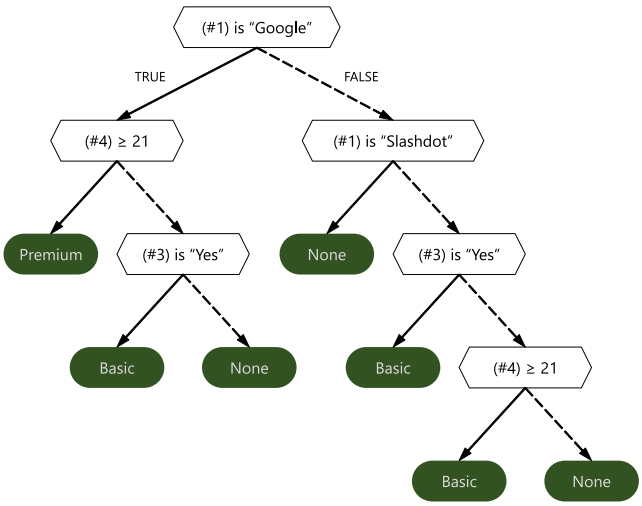

図 1　決定木モデルの例 [5]
Fig. 1　An example of a decision tree model

木モデルとその学習アルゴリズムについて説明する．次に 3 章で決定木学習を高速化する手法を提案し，4 章では提案手法と SX-Aurora TSUBASA による高速化の効果を評価する．5 章では決定木学習の並列化に関する関連研究を紹介し，最後に 6 章でまとめと今後の課題について述べる．

2. 決定木アルゴリズム

文献 [5] に記載の例を引用し，古典的な決定木アルゴリズムについて説明する．表 1 に学習データの例を，図 1 にこの学習データから構築される決定木モデルの例を示す．表 1 の学習データはひとつのデータ行がひとつの学習サンプルを，列が学習と推論に用いる特徴量および教師データに用いるラベルを示しており，この例は #1 から #4 までの四つの特徴量をもとに，None，Basic，Premium の三種類のクラスのいずれかひとつからなるラベルを推論するクラス分類タスクである．決定木を用いた推論では，ラベルが不明なデータ行を入力として，そのデータ行の特徴量に応じて決定木を根ノードから順に辿ることによって予測されるラベルを推論することができる．決定木の節ノードは，特徴量に関する条件判定とその結果によってどの枝を辿るべきかを示している．図 1 では条件判定結果が真のとき左の枝，偽のとき右の枝を辿ることを示している．条件判定では，特徴量 #1，#2，#3 のようなカテゴリ変数に対しては一致判定を，特徴量 #4 のような数値変数に対しては大小比較判定を行う．ひとつの節ノードが三つ以上の子ノードを持つような複雑な条件判定を行う決定木もあり得るが，本稿では一般的によく用いられる二分木形式のみを対象とする．最終的に葉ノードに至ると，その葉ノードに示されている値がデータ行に対する推論結果である．決定木には大きく分けて，非可算あるいは離散値で表されるクラスをラベルとして分類を行う分類木と，可算な連続値をラベルとして数値予測を行う回帰木の二種類があるが，いずれもほぼ同じ学習アルゴリズムで構築することができる．

決定木を構築するための中核となる処理は，入力となる学習データを重複も漏れもない部分学習データからなるグループに分割し，その分割の良し悪しを評価することである．学習データのうち i 行目の特徴量ベクトルを $\mathbf{x}_i$ で，i 行 j 列目の特徴量の値を x_{ij} で，i 行目のラベルを y_i で表すものとする．また，学習データ全体を $S = \{(\mathbf{x}_i, y_i)\}$ で，ふたつに分割された部分学習データをそれぞれ A, B で表すものとする．学習データを分割する基準は，条件判定に用いる特徴量の列番号 j，閾値 t，比較演算 cmp の三つを特定することにより定まり，式 (1) で表される．ただし，$A \neq \emptyset, B \neq \emptyset$ である．

$$\begin{aligned} A &= \{(\mathbf{x}_i, y_i) \in S \mid \mathrm{cmp}(x_{ij}, t) \text{ is true}\} \\ B &= S \setminus A \end{aligned} \tag{1}$$

分割の良し悪しは，確率分布の差異を計る尺度である情報利得（Information Gain）と呼ばれるスコアで評価し，そのスコア算出には学習データに含まれるラベルの不純度（Impurity）と呼ばれる値を用いる．学習データを入力としてラベルの不純度を出力する関数を f とすると，情報利得 G は式 (2) のように，分割前の学習データ S に対する不純度と，分割後の部分学習データ A, B それぞれに対する不純度の加重平均との差で表される．

$$G = f(S) - \frac{n_A f(A) + n_B f(B)}{n} \tag{2}$$

ここで n は学習データ S のデータ行数，n_A は A のデータ行数，n_B は B のデータ行数であり，$n = n_A + n_B$ である．決定木学習においてひとつの節ノードを構築するためには，入力の学習データ S に対して，式 (2) に示した情報

© 2020 Information Processing Society of Japan

利得 G が最大となるような分割の基準を探索する．既に述べたように，条件判定の比較演算 cmp としては特徴量がカテゴリ変数であれば一致判定，数値変数であれば大小比較判定を行うことになるため，分割基準の探索空間は特徴量の列番号 j および閾値 t の組である．ある (j,t) の組に対する情報利得を $G_{(j,t)}$ で表すと，最良の分割基準 (j^*,t^*) は式 (3) で表すことができる．

$$(j^*,t^*) = \arg\max_{(j,t)} G_{(j,t)} \tag{3}$$

なお，情報利得 G は最大値であってもゼロ以下の値となることがあり，その場合は学習データを適切に分割できないことを意味する．学習データを適切に分割できる最良の基準 (j^*,t^*) を発見できた場合はその基準 (j^*,t^*) を用いた条件判定に基づき学習データ S を分割し，一方の子ノードを構築するためには A を，もう一方の子ノードを構築するためには B をそれぞれ新たな入力学習データとして，同じ手順を再帰的に実行する．情報利得 G の最大値がゼロ以下となり学習データを分割できなくなると，当該ノードを葉ノードとして終端する．

ラベルの不純度を示す関数 f としてどのような計算方法を用いるべきかは，決定木が分類木か回帰木かによって異なる．分類木において不純度は学習データに含まれるラベルの種類と出現数によって定まり，ジニ不純度（Gini Impurity）や情報エントロピー（Information Entropy）と呼ばれる尺度が用いられる．n_s 行のデータ行からなる入力学習データ s に対し，分類対象のクラスが M 種類あるとき，k 種類目のクラス C_k が学習データ s のラベル列に含まれる個数を $c_{s,k}$ で表すと，ジニ不純度は式 (4) で，情報エントロピーは式 (5) で表される．

$$\mathrm{gini}(s) = \sum_{k=1}^{M} \frac{c_{s,k}}{n_s}\left(1 - \frac{c_{s,k}}{n_s}\right) \tag{4}$$

$$\mathrm{entropy}(s) = -\sum_{k=1}^{M} \frac{c_{s,k}}{n_s} \log \frac{c_{s,k}}{n_s} \tag{5}$$

一方，回帰木では目的が数値予測であることから，不純度には式 (6) で表される分散の値が用いられる．ただし $y_{s,i}$ は学習データ s に含まれる i 行目のラベル，μ_s は s に含まれるラベルの平均値である．

$$\mathrm{var}(s) = \frac{1}{n_s} \sum_{y_{s,i} \in s} \left(y_{s,i} - \mu_s\right)^2 \tag{6}$$

式 (2) に示したように，情報利得の計算には分割前の学習データに対する不純度 $f(S)$ だけではなく分割後の学習データに対する不純度 $f(A), f(B)$ の計算が必要となる．したがって式 (3) のように情報利得 $G_{(j,t)}$ が最大となる分割基準を探索するためには，列番号および閾値の組 (j,t) を様々な値に変化させた条件下で不純度を計算する必要がある．加えてジニ不純度および情報エントロピーの計算では，クラス C_k も条件に加わる．決定木学習のスコア評価はこのような複雑な条件の下でのデータ処理となり，ベクトルプロセッサを活用して学習を高速化するためには相応の工夫が必要となる．

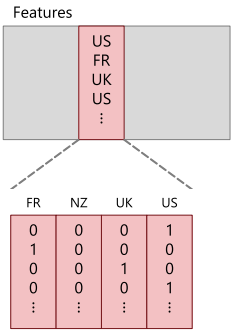

図 2　グループ分け行列
Fig. 2　A grouping matrix

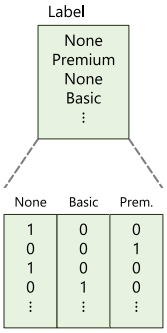

図 3　クラス分け行列
Fig. 3　A classification matrix

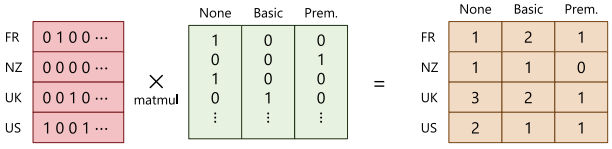

図 4　行列積による数え上げ
Fig. 4　Counting by matrix multiplication

3. 提案手法

3.1 分類木学習の高速化

本節ではベクトルプロセッサを活用して分類木の学習を高速化する手法として，多数の分割基準 (j,t) の条件に渡る不純度の算出を行列積演算で行う方式を提案する．具体的には，グループ分け行列 U およびクラス分け行列 V を中間データとして算出し，これらの行列積 $Z = {}^tUV$ を求めることでラベル列に含まれるクラスの個数を算出する．

ある j 列目の特徴量に対して様々な閾値で学習データを分割することを表現するグループ分け行列を $U^{(j)}$ とする．$U^{(j)}$ は 図 2 のようにひとつの分割基準 (j,t) 毎に一列をなし，それぞれの分割基準 (j,t) に対応する列の i 行目の要素を $u_i^{(j,t)}$ として，各要素 $u_i^{(j,t)}$ には式 (7) のように値を設定する．

$$u_i^{(j,t)} = \begin{cases} 1 & (\mathrm{cmp}(x_{ij}, t) \text{ is true}) \\ 0 & (\text{otherwise}) \end{cases} \tag{7}$$

このような個々の特徴量に対するグループ分け行列 $U^{(j)}$ を部分行列として，これらの部分行列を式 (8) のように横に並べた行列 U を全体のグループ分け行列とする．

$$U = \begin{bmatrix} U^{(1)} & U^{(2)} & U^{(3)} & \cdots \end{bmatrix} \tag{8}$$

次に，各学習データ行に付与されているラベル y_i によって学習データがクラス分けされることを表現する行列を V とする．V は 図 3 のようにひとつのクラス毎に一列をなし，k 種類目のクラス C_k に対応する列の i 行目の要素を

$v_i^{(k)}$ として，各要素 $v_i^{(k)}$ には式 (9) のように値を設定する．

$$v_i^{(k)} = \begin{cases} 1 & (y_i \text{ is } C_k) \\ 0 & (\text{otherwise}) \end{cases} \tag{9}$$

以上のようにグループ分け行列 U とクラス分け行列 V を定義すると，U の転置行列 ${}^tU$ と V との行列積 $Z = {}^tUV$ を算出することで，図 4 のように各分割基準 (j,t) で学習データを分割した際に各クラス C_k がラベル列に何個含まれるかを求めることができる．すなわち，分割基準 (j,t) かつクラス C_k に対応する Z の要素を $z_{(j,t),(k)}$ とし，U の各列ベクトルを $\mathbf{u}^{(j,t)}$，V の各列ベクトルを $\mathbf{v}^{(k)}$ とすると，$z_{(j,t),(k)}$ の値は式 (10) のように求められる．ただし n は分割前の学習データ行数である．

$$z_{(j,t),(k)} = \langle \mathbf{u}^{(j,t)}, \mathbf{v}^{(k)} \rangle = \sum_{i=1}^{n} u_i^{(j,t)} v_i^{(k)} \tag{10}$$

分割基準 (j,t) で式 (1) の通り分割したふたつの部分学習データを A, B とすると，A, B それぞれのラベル列に含まれるクラス C_k の個数 $c_{A,k}^{(j,t)}, c_{B,k}^{(j,t)}$ は式 (11) のように求められる．さらに，クラスが全部で M 種類あるとすると，分割後のデータ行数 $n_A^{(j,t)}, n_B^{(j,t)}$ は式 (12) のように求められる．

$$c_{A,k}^{(j,t)} = z_{(j,t),(k)}, \qquad c_{B,k}^{(j,t)} = \sum_{i=1}^{n} v_i^{(k)} - z_{(j,t),(k)} \tag{11}$$

$$n_A^{(j,t)} = \sum_{k=1}^{M} c_{A,k}^{(j,t)}, \qquad n_B^{(j,t)} = n - n_A^{(j,t)} \tag{12}$$

以上で分類木の不純度を求めるために必要な値が求められる．このようにデータの数え上げを行列積演算で代替することにより条件分岐の発生を回避することができ，ベクトルプロセッサを活用して様々な分割基準に対する不純度の算出を高速に行うことができる．

3.2 回帰木学習の高速化

回帰木に対しても，分類木と同様の行列演算の考え方を適用することができる．回帰木学習における提案手法の方針は，ラベル列ベクトル $\mathbf{y}$ との行列ベクトル積 $\mathbf{z} = {}^tU\mathbf{y}$ を求めることでラベル値の部分和を算出することである．

古典的な回帰木ではまず式 (6) により分散を，次に式 (2) により情報利得を求めることになるが，式 (13) で表される Friedman の最小二乗改善基準 [6] を利用すると計算量を減少させることができる．式 (2) の f に式 (6) の分散を代入して式変形を行うことで，式 (14) のように情報利得を直接求めることができる．

$$i^2(A,B) = \frac{n_A n_B}{n_A + n_B} (\mu_A - \mu_B)^2 \tag{13}$$

$$G = \frac{i^2(A,B)}{n} = \frac{n_A n_B}{n^2} (\mu_A - \mu_B)^2 \tag{14}$$

ここで n は分割前の学習データ行数，A, B は式 (1) の通り分割したふたつの部分学習データ，n_A, n_B はそれぞれ A, B の学習データ行数，μ_A, μ_B はそれぞれ A, B のみに含まれるラベルの平均値である．式 (14) により情報利得を直接求める場合でも，分割基準 (j,t) で学習データを A, B のふたつに分割した場合のデータ行数 $n_A^{(j,t)}, n_B^{(j,t)}$ およびラベルの平均値 $\mu_A^{(j,t)}, \mu_B^{(j,t)}$ が必要である．

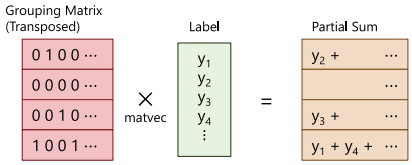

図 5 行列ベクトル積による部分和算出

Fig. 5 Partial summation by matrix-vector multiplication

回帰木の場合も学習データの特徴量部分の構成は分類木の場合と同じであるため，グループ分け行列 U を式 (7) および式 (8) と同様にして定義する．この U の各要素を $u_i^{(j,t)}$ とすると，データ行数 $n_A^{(j,t)}, n_B^{(j,t)}$ は式 (15) のように求めることができる．

$$n_A^{(j,t)} = \sum_{i=1}^{n} u_i^{(j,t)}, \qquad n_B^{(j,t)} = n - n_A^{(j,t)} \tag{15}$$

ラベル列ベクトル $\mathbf{y} = {}^t[y_1, \cdots, y_n]$ を用いると，グループ分け行列の転置行列 ${}^tU$ との行列ベクトル積 $\mathbf{z} = {}^tU\mathbf{y}$ を算出することで，図 5 のように各学習基準 (j,t) で学習データを分割した際のラベル値の部分和を求めることができる．すなわち，分割基準 (j,t) に対応する $\mathbf{z}$ の要素を $z_{(j,t)}$ とし，U の各列ベクトルを $\mathbf{u}^{(j,t)}$ とすると，$z_{(j,t)}$ の値は式 (16) のように求められる．

$$z_{(j,t)} = \langle \mathbf{u}^{(j,t)}, \mathbf{y} \rangle = \sum_{i=1}^{n} u_i^{(j,t)} y_i \tag{16}$$

以上の値を用いることにより，部分学習データのラベルの平均値 $\mu_A^{(j,t)}, \mu_B^{(j,t)}$ は式 (17) のように求めることができる．

$$\mu_A^{(j,t)} = \frac{z_{(j,t)}}{n_A^{(j,t)}}, \qquad \mu_B^{(j,t)} = \frac{1}{n_B^{(j,t)}} \left(\sum_{i=1}^{n} y_i - z_{(j,t)} \right) \tag{17}$$

このように部分和の算出を行列ベクトル積で代替することにより，ラベル y_i が部分学習データに含まれているか否かによる条件分岐の発生を回避することができ，ベクトルプロセッサを活用して様々な分割基準に対する情報利得の算出を高速に行うことができる．

3.3 行列のブロック化とビットベクトル化

大規模な学習データの場合，グループ分け行列 U が巨大なものとなり計算機のメモリに入りきらない問題が発生する．この問題は，縦横を適切なサイズに区切った部分行列により行列積を算出するブロック化を行うことで回避することができる．キャッシュ参照の局所性を高め処理を高速

© 2020 Information Processing Society of Japan

化する目的での行列積演算のブロック化は広く行われているが，提案手法においてはグループ分け行列 U の部分行列を必要な箇所のみ随時生成しながら行列積を算出していくことになる．この場合も副次的効果として U の部分行列を生成する際にキャッシュ参照の局所性が高まることが期待できるため，部分行列の大きさはメモリの最大サイズではなくキャッシュサイズを基準に調整する必要がある．

また，グループ分け行列 U およびクラス分け行列 V は全ての要素が 0 または 1 からなる行列である．そのためひとつの変数領域にひとつの行列要素を格納するのではなく，0 と 1 の並びをビットベクトルとみなして，ビット幅 N の整数型の変数領域ひとつに N 個の行列要素をまとめて格納することでメモリ領域を節約することができる．さらに，分類木ではポップカウント処理を利用して U, V をビットベクトル化した整数のまま行列積を算出することができる．ポップカウント処理とはビット列に対して 1 となっているビットの個数を数える処理であり，分割統治法などに基づく高速な実装 [7] が知られているほか，プロセッサによってはポップカウント処理を単一の命令で極めて高速に実行できるものがある．U, V の各要素をビット幅 N でビットベクトル化した整数をそれぞれ $\tilde{u}_i^{(j,t)}, \tilde{v}_i^{(k)}$ とし，ポップカウント関数を P とすると，式 (10) は式 (18) のように書き換えられる．ただし演算子 & はビット単位の論理積である．

$$z_{(j,t),(k)} = \sum_{i=1}^{\lceil n/N \rceil} P\left(\tilde{u}_i^{(j,t)} \,\&\, \tilde{v}_i^{(k)}\right) \tag{18}$$

しかしグループ分け行列 U をブロック化して随時生成しながら行列積を算出する場合には，ビットベクトルのまま行列積を算出することで削減できる演算時間と U をビットベクトルに変換する処理自体にかかる時間とが見合わず，U のビットベクトル化による高速化の効果が得られないことがある．一方，グループ分け行列 U をブロック化する場合にはクラス分け行列 V は一度生成したものを何度も参照することになるため，ビットベクトル化による高速化の効果が得やすい．そのため V が U ほどには巨大な行列にならないという条件付きではあるが，V の列数が多いほど，すなわち分類対象のクラスの種類数 M が多いほどビットベクトル化により処理を高速化することができる．

4. 性能評価

本章では，NEC データサイエンス研究所で開発している並列分散処理ミドルウェア Frovedis [8, 9] において提案手法を用いて実装した決定木の性能評価について説明する．比較対象は，x86 CPU 向け並列分散処理ミドルウェアである Apache Spark[*2] [10] が提供する機械学習ライブラリ MLlib に含まれる決定木とする．Frovedis は Spark に類似

[*2] https://spark.apache.org/

表 2 性能評価環境
Table 2 The evaluation environment

	SX-Aurora TSUBASA（A300）	
	Xeon Gold 6126 (Skylake)	Vector Engine Type 10B
コア仕様		
動作周波数	2.6 GHz	1.4 GHz
理論演算性能 ⋆	224.0 GFLOPS	537.6 GFLOPS
平均メモリ帯域	10.6 GB/s	153.6 GB/s
プロセッサ仕様		
コア数	12 /socket	8 /socket
理論演算性能 ⋆	1.76 TFLOPS	4.30 TFLOPS
最大メモリ帯域	0.12 TB/s	1.22 TB/s
キャッシュ容量	19.25 MB	16.00 MB
メモリ容量	192 GB ⋆⋆	48 GB
ソフトウェア		
Spark	Spark 2.4.0	–
Java	OpenJDK 1.8.0	–
C++	GCC 4.8.5	NCC 2.5.1
行列演算	OpenBLAS 0.3.5	NEC NLC 2.0.0
MPI	Open MPI 3.0.0	NEC MPI 2.3.0
VE 制御	–	VEOS 2.2.2

⋆ 単精度，⋆⋆ ホストメモリ

表 3 性能評価用の学習データセット
Table 3 Datasets for performance evaluation

データセット名	タスク	行数	列数	サイズ ⋆
KDD Cup 1999 [*4]	分類	4.90 M	125	2.26 GB
Molecular Properties [*5]	回帰	4.66 M	312	5.40 GB

⋆ 単精度浮動小数点

したインターフェースを持ちつつ内部は C++ および MPI を用いて実装されており，C++ API に加えて Spark ライクな Scala API や scikit-learn[*3] ライクな Python API を提供することで，SX-Aurora TSUBASA を用いた高速な処理を手軽に利用可能なミドルウェアである．提案手法を用いた決定木も Frovedis 内の機械学習ライブラリのひとつとして実装している．

SX-Aurora TSUBASA は x86 CPU として Intel Xeon を，ベクトルプロセッサを内蔵したアクセラレータとしてベクトルエンジン（Vector Engine，以下 VE）を搭載しており，Frovedis も x86 CPU と VE の両アーキテクチャに対応している．性能評価に用いたハードウェアおよびソフトウェアの環境を 表 2 に，学習用データセットを 表 3 に示す．分類木の学習には 5 クラス分類タスクである KDD Cup 1999[*4] の Full Dataset を用いており，特徴量エンジニアリングはカテゴリ変数を One-hot Encoding するのみの簡易なものとした．また，回帰木の学習には数値予測タスクである Predicting Molecular Properties（CHAMPS

[*3] https://scikit-learn.org/
[*4] https://archive.ics.uci.edu/ml/datasets/KDD+Cup+1999+Data

© 2020 Information Processing Society of Japan

表 4　分類木の学習時間
Table 4　Training time for a classification tree

コア数	学習時間（秒）		
	Spark (x86)	提案手法 (x86)	提案手法 (VE)
1	139.99	48.36	4.49
2	78.80	28.18	3.60
4	45.35	17.67	1.84
8	26.16	11.64	1.33
12	25.90	10.46	–

表 5　分類木の学習時間比
Table 5　Acceleration ratio for a classification tree

コア数	Spark（x86） / 提案手法（x86）	提案手法（x86） / 提案手法（VE）	Spark（x86） / 提案手法（VE）
1 vs. 1	2.89	10.76	31.16
2 vs. 2	2.80	7.82	21.87
4 vs. 4	2.57	9.59	24.62
8 vs. 8	2.25	8.73	19.61
12 vs. 12	2.48	–	–
12 vs. 8	–	7.84	19.42

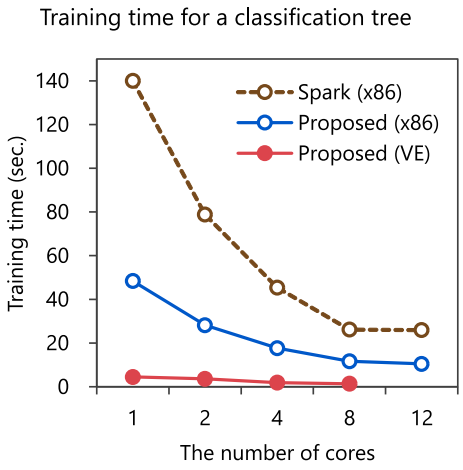

図 6　分類木の学習時間
Fig. 6　Training time for a classification tree

表 6　回帰木の学習時間
Table 6　Training time for a regression tree

コア数	学習時間（秒）		
	Spark (x86)	提案手法 (x86)	提案手法 (VE)
1	285.57	935.28	18.32
2	145.06	475.93	10.12
4	79.64	249.88	6.20
8	43.74	129.98	4.86
12	34.74	95.19	–

表 7　回帰木の学習時間比
Table 7　Acceleration ratio for a regression tree

コア数	Spark（x86） / 提案手法（x86）	提案手法（x86） / 提案手法（VE）	Spark（x86） / 提案手法（VE）
1 vs. 1	0.31	51.06	15.59
2 vs. 2	0.30	47.02	14.33
4 vs. 4	0.32	40.27	12.83
8 vs. 8	0.34	26.72	8.99
12 vs. 12	0.36	–	–
12 vs. 8	–	19.57	7.14

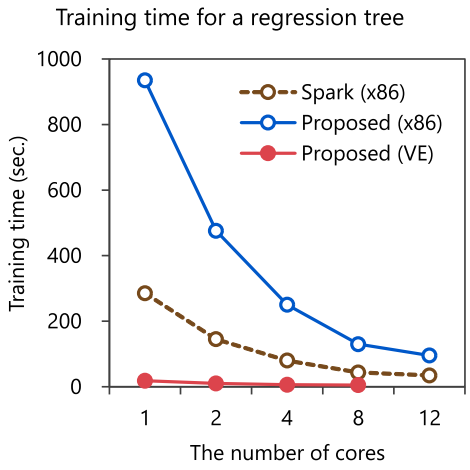

図 7　回帰木の学習時間
Fig. 7　Training time for a regression tree

Scalar Couping Challenge）[*5] のデータを用いており，特徴量エンジニアリングはオープンソースとして公開されているノートブック[*6] を使用した．なお，学習にかかる時間には学習データセットの読み込みにかかる I/O 時間は含めずに測定するものとする．構築する決定木の高さは Spark，提案手法ともに最大で 5 までとし，提案手法において 3.3 節で述べたグループ分け行列の部分行列化については，簡易的なチューニングの結果，一度に生成する部分行列のサイズをコア間共有で最大 512 MB までとしている．

以下，x86 CPU 上で Spark を用いた場合，x86 CPU 上で Frovedis を用いた場合，VE 上で Frovedis を用いた場合の三つの場合についてそれぞれ性能評価を行った結果を説明する．分類木の学習にかかる時間を 表 4 および 図 6 に，比較対象ごとの学習時間の比を 表 5 に示す．ただし，今回の性能評価に用いた VE は 1 ソケットあたりのコア

*5 https://www.kaggle.com/c/champs-scalar-coupling
*6 https://www.kaggle.com/felipemello/features-for-top-5-lb-with-nn-or-lgb

© 2020 Information Processing Society of Japan

数が 8 であるため，x86 CPU 12 コアとの学習時間比の算出には VE 8 コアの場合の学習時間を用いている．これは 1 ソケット同士の比較とみなすことができる．表 5 の結果から，x86 CPU 上で Spark に対して提案手法で 2.5 倍前後，x86 CPU 上での提案手法に対して VE 上での提案手法で 8 倍前後，全体として Spark 比で約 20 倍の高速化ができていることがわかる．次に，回帰木の学習にかかる時間を **表 6** および **図 7** に，比較対象ごとの学習時間の比を **表 7** に示す．回帰木では残念ながら x86 CPU 上では Spark より提案手法のほうが遅い結果となっており，回帰木に特化したさらなる改良の余地があることがうかがえる．ただし，x86 CPU 上での Spark と VE 上での提案手法との比較では約 7 倍の高速化ができていることがわかり，提案手法とベクトルプロセッサの相性は良いと言える．

5. 関連研究

学習データの並列分散処理による決定木の高速化に関する研究には，古くは Kufrin [11] や久保田ら [12] によるものなどがある．Kufrin [11] は各計算ノードでローカルに処理した途中結果を順番に他の計算ノードと交換していくことで最良の分割基準を探索する手法を用いている．また久保田ら [12] はいくつかの並列処理の方式がある中で，学習データ行を分散する場合，特徴量列を分散する場合，計算ノード内のスレッド並列化を行う場合，計算ノード間の並列化を行う場合に，学習データの偏りが学習速度にどのような影響を及ぼすかを評価している．近年の大規模化した学習データに対応するための並列処理手法として Meng ら [13] は，各計算ノードでローカルに処理した途中結果を多数決により大胆に絞り込んで計算ノード間の通信を削減しながらも，最終的に最良に近い分割基準を発見できる確率を詳細に分析し，学習精度を可能な限り落とさずに学習速度を高速化する手法を用いている．

決定木をベースとしたアンサンブル学習では場合により数百から数千の決定木を構築する必要がある．ランダムフォレスト（Random Forest） [1; Chapter 15] では学習データをサンプリングして性格の異なる決定木を複数構築するが，これらの決定木は独立して学習することが可能なため，タスク並列方式で別々の計算ノードが別々の決定木学習を行う方法が有効である．一方 GBDT（Gradient Boosting Decision Tree） [1; Chapter 10] は回帰木をベースとするアンサンブル学習だが，GBDT 学習においては m 番目の回帰木の学習結果を用いて $m+1$ 番目の回帰木の学習を行うため，個々の回帰木は逐次的に構築するほかなく，高速化のためには回帰木学習そのものの並列化が求められる．このような理由から GBDT の高速化に関する研究では主に GPU を対象として SIMD 方式の並列化を行うものがある [14, 15]．Zhang ら [14] は GPU のアトミック操作の利用に際し，データ更新の衝突を減少させる手法を用いている．GBDT の実装として近年もっとも有名なものには XGBoost[*7] [3] や LightGBM[*8] [4] がある．より正確には XGBoost は GBDT を改良したアルゴリズムを，LightGBM は XGBoost をさらに改良したアルゴリズム用いているが，ベースとなる学習モデルは原則として GBDT と同じく回帰木である．Mitchell ら [15] は XGBoost において，学習データの分割の際にメモリ上の物理的なデータ配置を可能な限り動かさずに済むデータ構造を採用し，加えて GPU に特化した実装を用いることで高速化を図っている．

さらに，決定木の学習後の推論フェーズを高速化する試みも為されており，Nakandala ら [16] の Hummingbird[*9] は推論アルゴリズムを行列積演算で代替することで処理を高速化する手法を用いている．本稿の提案手法と一部類似する点があると言えるが，学習フェーズと推論フェーズの違いから，Nakandala らの手法は木が深くなるにつれ行列のサイズが爆発的に巨大化するという課題がある．

6. まとめと今後の課題

本稿では，SX-Aurora TSUBASA が搭載するベクトルエンジンを用いて決定木学習を高速に実行することができる手法を提案した．Apache Spark の機械学習ライブラリ MLlib との比較の結果，SX-Aurora TSUBASA のベクトルエンジンと提案手法を組み合わせた場合，分類木で約 20 倍，回帰木で約 7 倍の高速化が可能なことを確認した．4 章で紹介した並列分散処理ミドルウェア Frovedis はオープンソースソフトウェアとして GitHub に公開しており[*10]，提案手法も Frovedis 内の機械学習ライブラリのひとつとして公開済みである．

今後の課題としては回帰木および GBDT に特化した対応が挙げられる．提案手法は分類木に対して大きな高速化の効果をもたらしているが，回帰木に対してはまだ改良の余地があるとみられる．また，Kaggle に代表されるような実データを用いた機械学習のコンペティションで，XGboost や LightGBM などの GBDT 系アルゴリズムが活躍していることも GBDT に注力すべき背景と言える．XGBoost における Mitchell ら [15] の実装は GPU に特化しているものの，その手法の一部はベクトルプロセッサに対しても適用可能である．特に，メモリ上の学習データを初期配置から動かさない手法を採用する場合には，学習データの分割に伴ってコアからのメモリアクセスが複雑化するという特徴があるため，SX-Aurora TSUBASA のベクトルエンジンがもつ広いメモリ帯域が有効である可能性があり，今後検討を進める予定である．

*7 https://github.com/dmlc/xgboost
*8 https://github.com/microsoft/LightGBM
*9 https://github.com/microsoft/hummingbird
*10 https://github.com/frovedis

© 2020 Information Processing Society of Japan

参考文献

[1] Hastie, T., Tibshirani, R. and Friedman, J.: *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Springer, 2nd edition (2009).
[2] Zhou, Z. and Feng, J.: Deep Forest: Towards an Alternative to Deep Neural Networks, *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI-17)*, Melbourne, Australia (2017).
[3] Chen, T. and Guestrin, C.: XGBoost: A Scalable Tree Boosting System, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD '16)*, San Francisco, USA, pp. 785–794 (2016).
[4] Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q. and Liu, T.: LightGBM: A Hightly Efficient Gradient Boosting Decision Tree, *Proceedings of the 31st Annual Conference on Neural Information Processing Systems (NeurIPS 2017)*, Long Beach, USA (2017).
[5] Segaran, T. (當山 仁健, 鴨澤 眞夫 訳) : 集合知プログラミング, pp. 155–180, オライリー・ジャパン (2008).
[6] Friedman, J. H.: Greedy function approximation: A gradient boosting machine, *Annals of Statistics*, Vol. 29, No. 5, pp. 1189–1232 (2001).
[7] Hamming weight, Wikipedia (online), available from ⟨https://en.wikipedia.org/wiki/Hamming_weight⟩ (accessed 29th Jun. 2020).
[8] Araki, T.: Accelerating Machine Learning on Sparse Datasets with a Distributed Memory Vector Architecture, *Proceedings of the 16th International Symposium on Parallel and Distributed Computing (ISPDC 2017)*, Innsbruck, Austria (2017).
[9] 荒木 拓也, 大野 善之, 石坂 一久：ベクトルプロセッサを用いた AI 処理の高速化, 電子情報通信学会誌, Vol. 103, No. 5, pp. 529–534 (2020).
[10] 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 Symposium on Networked Systems Design and Implementation (NSDI '12)*, San Jose, USA, pp. 15–28 (2012).
[11] Kufrin, R.: Decision Trees on Parallel Processors, *Machine Intelligence and Pattern Recognition*, Vol. 20, pp. 279–306 (1997).
[12] 久保田 和人, 仲瀬 明彦, 酒井 浩, 小柳 滋：決定木の並列化とその評価, 情報処理学会研究報告, 1999-HPC-77, Vol. 1999, No. 66, pp. 161–166 (1999).
[13] Meng, Q., Ke, G., Wang, T., Chen, W., Ye, Q., Ma, Z. and Liu, T.: A Communication-Efficient Parallel Algorithm for Decision Tree, *Proceedings of the 30th Annual Conference on Neural Information Processing Systems (NeurIPS 2016)*, Barcelona, Spain (2016).
[14] Zhang, H., Si, S. and Hsieh, C.: GPU-acceleration for Large-scale Tree Boosting, *arXiv.org*, (online), available from ⟨https://arxiv.org/abs/1706.08359⟩ (2017).
[15] Mitchell, R. and Frank, E.: Accelerating the XGBoost algorithm using GPU computing, *PeerJ Computer Science*, (online), available from ⟨https://peerj.com/articles/cs-127/⟩ (2017).
[16] Nakandala, S., Yu, G., Weimer, M. and Interlandi, M.: Compiling Classical ML Pipelines into Tensor Computations for One-size-fits-all Prediction Serving, *Workshop on Systems for ML at NeurIPS 2019*, Vancouver, Canada (2019).

© 2020 Information Processing Society of Japan