データマイニングのアルゴリズム記述を容易にする拡張行列演算の提案

松田一孝[†] 筧一彦[†]
胡振江[†,††] 武市正人[†]

計算機性能の向上およびそれに伴うデータ量の増加により，知識発見，データマイニングの重要性が高まってきている．その処理には多くのメモリ，高い演算能力が必要であり並列分散化の必要があり，個々の問題に対しては様々な並列分散化が施されている．しかし，一般に並列分散アルゴリズムの記述を行うことは難しい．

本研究では，データマイニングにおける並列分散アルゴリズムの記述を容易にすることを目的として，「拡張行列演算」と呼ぶ並列アルゴリズム記述の枠組みを提案する．この枠組みは行列演算の加算および乗算の演算子を一般化したものである．枠組みの持つ計算パターンは少ないが，行列演算のアナロジーがアルゴリズム記述を容易かつ簡潔にしており，また問題に対し適切な演算子を定義することができるためそのその表現力は高い．行列計算の並列化に関する研究は多く，同様にこの枠組みの分散並列化も行うことができる．計算パターンが少ないことはプログラムの代数的な取り扱いが容易となる利点も持つ．

本論文では，枠組みの概要，いくつかのデータマイニングのアルゴリズム記述例を示し，実験を通してこの枠組みの有用性を示す．

Extended Matrix Operations for describing Data Mining Algorithms

Kazutaka Matsuda,[†] Kazuhiko Kakehi,[†] Zhenjiang Hu[†,††]
and Masato Takeichi[†]

The increase of machine power and the existence of the concomitant huge-sized database have made knowledge discovery and data mining possible and more important. Processing such massive date requires huge computational power and memory as well, which calls for distributed and parallel treatments. Although there have been many case studies of parallizing data mining algorithms in ad hoc manners, describing parallel and distributed data mining algorithms is still a hard task.

In this paper, we propose a framework, called *extended matrix operations*, for describinig parallel and distributed data mining algorithms in a general and uniform way. This framework is a generalization of matrix operations whose operators of addition and multiplication are generalized. This framework has the following advantages: Analogy to usual matrix operations makes intuitive and concise description of algorithms; user can implement many algorithms through giving proper definitions of the generalized operators; limited number of computation patterns makes algebraic treatments of programs easy. Matrix operations have a large number of researchs on parallelization, which also apply to our framework. We explain the framework and demonstrate how concisely data miming algorithms are described in our framework. Effectiveness of our framework is examined by experiments.

1. はじめに

計算機性能の向上，そして商業の情報化に従い，大量のデータがデータベースに蓄えられきた．そのような大量のデータから意味のある情報を見つけだす，知識発見，データマイニングの手法は重要性を増してきている．膨大なデータを高速に処理するためには，大きなメモリ領域や，高い演算能力が同時に必要であり，データマイニングのアルゴリズムの並列分散化が必要となる．

こうした中，個々のアルゴリズムについては，並列化のための多くの研究が存在している[8)11)]．しかし，それらの手法は問題ごとに提供されるアドホックなものが多く，他の問題に適用するのは容易ではない．このように，一般にデータマイニングのアルゴリズムを

† 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology , University of Tokyo
†† 科学技術振興機構 さきがけ研究 21
PRESTO 21, Japan Science and Technology Agency

並列分散で記述することは難しく，新たなデータマイニングのアルゴリズムの設計は逐次と並列化とをそれぞれ行う必要がある．

データマイニングで扱われるデータ構造は，関係データベースに見られるように二次元のものが多く用いられる．そこで本論文では二次元配列と呼ばれる行列状のデータ構造を対象とする．リストや二分木といった再帰的に構成されるデータ構造の場合，その構成方法に沿った自然で直観的なデータの走査方法が考えられ，そのデータ構造を用いたアルゴリズムの多くはその走査方法の上で記述できることが知られている[5]．この基本となる走査方法に対する並列分散計算法を示せば，多くのアルゴリズムが並列実行可能となる．二次元配列というデータ構造についても，様々な再帰的な構成法が提案されており，その並列実行方法について研究が行われている[3),13),20)]．しかしながら，こうした構成法は二次元配列上のアルゴリズムを記述する方法としては直観的ではない．

本論文ではまず走査に着目する．二次元配列の基本的な走査として，行列の演算のアナロジーを考える．行列の演算はその走査方法が単純であり，直観的な理解や記述がしやすい上に，数学的な取り扱いがしやすい．また，行列は工学上重要なためその計算方法の並列分散化に関して様々な研究が存在している[10]．本論文では，行列の演算を拡張した走査を「拡張行列演算」として提案し，この枠組みの並列実行法，またその枠組みの上でプログラミングを行える環境の実現を示す．拡張行列演算では，行列加算，行列乗算，行列スカラー積の 3 つに対して，演算子 $\times$，$+$ を一般化し，関数 $\otimes$，$\oplus$ へ拡張した計算パターンが構成要素となっている．これは，二次元配列上の走査を直観的に理解しやすく扱いやすいものに限定することを意味する．しかし，行列演算のアナロジーで与えられるような自然な走査のみに限ったとしても，後の説で述べる様に様々な関数を記述することができる．

我々の提案する拡張行列演算の特長は以下の通りである．

簡潔な記述 拡張行列演算は，行列演算のアナロジーとして提供される．そのため，直観的な記述が行いやすく，また計算パターンが拡張行列加算，拡張行列乗算，拡張行列スカラー積の 3 つと簡潔である．実際に計算に用いられる関数は計算パターンと独立しており，計算パターンの再利用性が高い．並列プログラムで一般に正しく記述するのが難しいとされる通信，同期は拡張行列演算の下では隠蔽する．

高い表現力 拡張行列演算は強力であり，データマイニングでの様々なアルゴリズムを記述することができる．たとえば，連想計算と呼ばれるテキストマイニングエンジンである GETA[9] での主な計算は，拡張行列演算を用いて表現することが可能である．

並列分散実行 行列演算における基本的な走査について，既に並列分散手法が知られており[22)]，拡張行列演算についても同様に並列分散化できる．そのため，拡張行列演算を用いてアルゴリズムを記述することで，アルゴリズムが並列分散で実行できることが保証される．

代数変換による効率化 プログラムの効率を向上させる手法として，プログラムを書き換えるプログラム変換が知られている[17]．拡張行列演算によるプログラミングは計算パターンを組み合わせにることによって行われるため，その計算パターンの組み合わせ方や計算パターンが引数としてとる演算子の持つ性質を代数的に取り扱うことでプログラムの導出が行える．

我々はこの拡張行列演算を利用するための環境を実現した．ユーザは関数型言語の豊富な表現力を利用して，様々なアルゴリズムを自然に記述することができ，なおかつ，並列分散環境下で実行できる．我々はさらに，データマイニングのアルゴリズムの記述を行い，実験により拡張行列演算の有効性を示す．

本論文は以下のように構成される．第 2 節では，本論文で使用される表記法について説明する．第 3 節では，本研究の提案する二次元配列上の拡張行列演算について述べる．第 4 節では拡張行列演算の表現力を示すため，実際に拡張行列演算を用いていくつかのデータマイニングの問題を記述する．第 5 節では，拡張行列演算の並列分散化について議論する．第 6 節では，拡張行列演算の実装を示す．第 7 節では，この拡張行列演算で記述されたプログラムのプログラム変換について述べる．第 8.2 節で本研究の関連性の高い研究を紹介する．第 8 節で本論文の内容をまとめ，今後の課題を示す．

2. 表　　記

本論文では関数型言語 Haskell[4] に倣った記法を用いる．

2.1 関 数 定 義

関数定義は次のように書く．

$$double\ x = x + x$$

これは，引数を 2 倍する関数 *double* の定義である．引

$$
\begin{aligned}
&([\cdot])\ a = [a]\\
&[] +\!\!+ bs = bs\\
&(a : as) +\!\!+ bs = a : (as +\!\!+ bs)\\
&\mathsf{map_L}\ f\ [] = []\\
&\mathsf{map_L}\ f\ (a : as) = f\ a : \mathsf{map_L}\ f\ as\\
&\mathsf{zipWith_L}\ f\ []\ [] = []\\
&\mathsf{zipWith_L}\ f\ (a : as)\ (b : bs) = f\ a\ b : \mathsf{zipWith_L}\ f\ as\ bs\\
&\mathsf{foldr_L}\ (\oplus)\ e\ [] = e\\
&\mathsf{foldr_L}\ (\oplus)\ e\ (a : as) = a \oplus \mathsf{foldr_L}\ (\oplus)\ e\ as
\end{aligned}
$$

図 1 List を扱う関数
Fig. 1 Utility functions for list

数の括弧は省略され，スペースが用いられる．関数適用も同様にスペースを用いて表し，括弧は省略する．関数はカリー化され，関数適用は左結合である．すなわち，$f\ a\ b$ は，$f\ (a\ b)$ でなく $(f\ a)\ b$ を表す．例として，2 つの値をとりその和を返す，カリー化された 2 引数関数 *add* は以下のように定義される．

$$add\ x\ y = x + y$$

ここで，引数に 1 増やした値を返す *inc* は

$$inc = add\ 1$$

のように定義できる．*add* は引数を 1 つとり，「引数を 1 つとって値を返す関数」を返す関数であると考えられる．また，関数適用が最も結合順位が高い．よって $double\ 1 + 2$ の返り値は $double\ (1 + 2) = 6$ ではなく，$(double\ 1) + 2 = 4$ である．また中置二項演算子 $\oplus$ は $(\oplus)$ と表記することで，

$$a \oplus b = (\oplus)\ a\ b$$

のように 2 引数関数を表すことができる．これをセクション化という．さらに，引数を 1 つだけセクション化して，以下のように 1 引数関数にできる．

$$a \oplus b = (a\oplus)\ b = (\oplus b)\ a$$

たとえば，*inc* は，

$$inc = (1+)$$

とセクションを用いても定義できる．

関数型言語では，関数を再帰的に定義することができる．たとえば，フィボナッチ数列の n 番目を計算する関数 *fib* は以下のように定義される．

$$
\begin{aligned}
&fib\ 0 = 1\\
&fib\ 1 = 1\\
&fib\ n = fib\ (n-1) + fib\ (n-2)
\end{aligned}
$$

2.2 組

値を (a, b) や，(x, y, z) のようにいくつかまとめたものを組と呼ぶ．組 $(a_1, \ldots, a_n)$ に対して $a_1, \ldots, a_n$ の型は全て違っていてもよい．

組を利用して計算量が $\mathrm{O}(n)$ の *fib* を定義できる．

$$
\begin{aligned}
&fib\ n = u\\
&\quad \textbf{where}\quad (u, v) = fib'\ n\\
&fib'\ 0 = (1, 0)\\
&fib'\ n = (u + v, u)\\
&\quad \textbf{where}\quad (u, v) = fib'\ (n-1)
\end{aligned}
$$

上記のように，関数の定義に必要な値は適宜 **where** 節の中で束縛される．

2.3 リ ス ト

リストは，空リストを表す []，リストと値を取りリストを作る : という 2 つのデータ構成子からなる均質なデータ構造である．データ構成子 : は右結合である．簡便のため $1 : 2 : 3 : []$ を [1,2,3] と書く．$[1, (2, 3)]$ などは，1 と $(2, 3)$ の型が異なり均質でないので許されない．

リストに対する基本的な関数の例を図 1 に示す．$([\cdot])$ は単一要素リストの生成，$+\!\!+$ はリスト同士の連接，$\mathsf{map_L}\ f$ は全ての要素に対する関数 f の適用，$\mathsf{zipWith_L}$ は 2 つのリストの綴じ合わせ，そして $\mathsf{foldr_L}$ はリストを 1 つの値に縮約する関数である．いずれの再帰関数もリストの構成のされ方に従って定義されており，このうち $\mathsf{foldr_L}$ がリストに対する基本的な走査方法を提供するものとなっている．これらの再帰関数はデータ構造の走査方法のみを提供している．実際の計算は，引数となる関数の性質に依存している．

2.4 二次元配列

二次元配列とは，

$$
A = \begin{pmatrix}
a_{00} & \cdots & a_{0(m-1)}\\
\vdots & \ddots & \vdots\\
a_{(n-1)0} & \cdots & a_{(n-1)(m-1)}
\end{pmatrix}
$$

のような均質な $n \times m$ 個の要素を持つデータ構造である[4]．

本論文で $n \times m$ 二次元配列と書いた場合，サイズが $n \times m$ である二次元配列を指す．原則として，二次元配列の表記には A のように全て大文字の文字列を使い，二次元配列に格納された要素を差し示すには 2 つのインデックス $i \in \{0, \ldots, n-1\}$，$j \in \{0, \ldots, m-1\}$ を用いて，a_{ij} のように対応する全て小文字の文字列にインデックスを下に付加したものを用いる．

$n \times m$ 二次元配列 A に対して，サイズが $m \times n$ であり，$b_{ji} = a_{ij}$ である二次元配列を考えることができる．このとき B を A の転置であるといい A^T と表記する．$(A^T)^T = A$ である．$A = B$ と書いたとき，A と B は同じサイズで全ての要素の値が等しいことを表す．

関数を定義する際，引数にどのような値が与えられても計算結果に影響がない場合がある．Ω は計算に影響を及ぼさない二次元配列の引数を表すものとする．

二次元配列を表現するデータ構造としては，様々なものが挙げられる[1),13),20)]．本論文では，コンピュータ内で二次元配列が実際にどのように定義され，実装されているかには言及しない．

2.5 型

本論文内では，要素の型は $\mathcal{A}, \mathcal{B}, \ldots$ のように表記する．組 $(a_1, \ldots, a_n)$ の持つ型は，a_i の型を $\mathcal{A}_i$ とすると，$(\mathcal{A}_1 \times \cdots \times \mathcal{A}_n)$ であらわす．また，要素の型が $\mathcal{A}$ であるリストの型を $list\ \mathcal{A}$ と書く．さらに，要素の型が $\mathcal{B}$ である二次元配列の型を $matrix\ \mathcal{B}$ と書く．

型が $\mathcal{A}$ である引数を 1 つとり $\mathcal{B}$ の型の値を返す関数の型は $\mathcal{A} \to \mathcal{B}$ と書かれる．同様に，引数を n 個取る関数では，引数の型を $\mathcal{A}_1, \ldots, \mathcal{A}_n$, 返り値の型を $\mathcal{B}$ とした場合，カリー化により $\mathcal{A}_1 \to \cdots \to \mathcal{A}_n \to \mathcal{B}$ と書く．$\to$ は右結合的である．たとえば，add の型は，$\mathbb{R} \to \mathbb{R} \to \mathbb{R}$ である．1 つ引数をとった $add\ 1$ の型は $\mathbb{R} \to \mathbb{R}$ となる．

3. 拡張行列演算

我々は，拡張行列演算という新たな枠組みを定義する．拡張行列演算は，行列乗算の拡張である拡張行列乗算，行列スカラー積の拡張である拡張行列スカラー積，行列加算の拡張である拡張行列加算の 3 つから成る．

3.1 並列スケルトン

拡張行列演算では，適切な演算子が与えられた計算パターンを組み合わせることでプログラムが記述される．このように並列計算できる計算パターンを用いてプログラムを行う手法をスケルトン並列プログラミングといい，その計算パターンを並列スケルトンという[7)]．拡張行列演算での具体的な計算パターンを説明する前に，関連研究として二次元配列上の並列スケルトン[15)] の中で本研究と関連の深い $\mathsf{map}_{\mathrm{M}}$，$\mathsf{zipWith}_{\mathrm{M}}$，$\mathsf{reduceCol}_{\mathrm{M}}$，$\mathsf{reduceRow}_{\mathrm{M}}$ を紹介する．

$\mathsf{map}_{\mathrm{M}}\ f$ は，二次元配列の全要素に関数 f を適用する関数である．たとえば

$$\mathsf{map}_{\mathrm{M}}\ f \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} f\,a & f\,b \\ f\,c & f\,d \end{pmatrix}$$

となる．これは拡張行列スカラー積と関連が深い．

$\mathsf{zipWith}_{\mathrm{M}}\ f$ は，2 つの二次元配列を関数 f を用いて綴じ合せる関数である．たとえば

$$\mathsf{zipWith}_{\mathrm{M}}\ f \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x & y \\ z & w \end{pmatrix} = \begin{pmatrix} f\,a\,x & f\,b\,y \\ f\,c\,z & f\,d\,w \end{pmatrix}$$

となる．これは拡張行列加算と関連が深い．

そして，$\mathsf{reduceRow}_{\mathrm{M}}$ は行方向に，$\mathsf{reduceCol}_{\mathrm{M}}$ は列方向に，結合的な演算子 $\oplus$ で縮約する関数である．たとえば

$$\mathsf{reduceRow}_{\mathrm{M}}\ (\oplus) \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} a \oplus b \\ c \oplus d \end{pmatrix}$$

であり，

$$\mathsf{reduceCol}_{\mathrm{M}}\ (\oplus) \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} a \oplus c & b \oplus d \end{pmatrix}$$

である．これは，拡張行列乗算を用いて記述することができる．

3.2 拡張行列乗算

拡張行列乗算は，2 つの二次元配列間に定義される演算の 1 つであり，行列乗算での演算子を一般化したものである．型が $\mathcal{A} \to \mathcal{B} \to \mathcal{C}$ である $\otimes$ と，型が $\mathcal{C} \to \mathcal{C} \to \mathcal{C}$ である結合的な $\oplus$ を用いて，次のように拡張行列乗算を定義する．

$$Z = X \left\langle\!\!\left\langle \begin{matrix} \otimes \\ \oplus \end{matrix} \right\rangle\!\!\right\rangle Y$$
$$\overset{\mathrm{def}}{\Longleftrightarrow} z_{ij} = \bigoplus_{0 \le k \le m-1} x_{ik} \otimes y_{kj}$$

但し，X は $n \times m$，Y は $m \times l$ の二次元配列であり，Z は $n \times l$ の二次元配列である．このとき X の型は $matrix\ \mathcal{A}$，Y の型は $matrix\ \mathcal{B}$，Z の型は $matrix\ \mathcal{C}$

である．たとえば

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}, \begin{pmatrix} s \\ t \end{pmatrix}$$

という 2 つの二次元配列に対して，拡張行列乗算は

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \left\langle\!\!\left\langle \begin{matrix} \otimes \\ \oplus \end{matrix} \right\rangle\!\!\right\rangle \begin{pmatrix} s \\ t \end{pmatrix} = \begin{pmatrix} (a \otimes s) \oplus (b \otimes t) \\ (c \otimes s) \oplus (d \otimes t) \end{pmatrix}$$

と計算される．拡張行列乗算をはじめとする拡張行列演算は行列走査の計算パターンを提供するのみであり，リストでの $\mathsf{foldr}_{\mathrm{L}}$ と同様に実際の計算は $\otimes, \oplus$ などの演算子に依存している．たとえば，

$$X \left\langle\!\!\left\langle \begin{matrix} \times \\ + \end{matrix} \right\rangle\!\!\right\rangle Y$$

は通常の行列乗算と一致する．このことから，拡張行列乗算が一般の行列乗算を拡張したものとなっていることがわかる．

二次元配列上の並列スケルトン $\mathsf{reduceRow}_{\mathrm{M}}$, $\mathsf{reduceCol}_{\mathrm{M}}$ は拡張行列演算の枠組みで記述することができる．たとえば，$\mathsf{reduceRow}_{\mathrm{M}}$ は縦ベクトル Ω に対し，

$$\mathsf{reduceRow}_{\mathrm{M}}\ (\oplus)\ X = X \left\langle\!\!\left\langle \begin{matrix} \ll \\ \oplus \end{matrix} \right\rangle\!\!\right\rangle \Omega \quad \textbf{where}\ \ x \ll y = x$$

と書ける．そして，$\mathsf{reduceCol}_{\mathrm{M}}$ は横ベクトル Ω に対し，

$$\mathsf{reduceCol}_{\mathrm{M}}\ (\oplus)\ X = \Omega \left\langle\!\!\left\langle \begin{matrix} \gg \\ \oplus \end{matrix} \right\rangle\!\!\right\rangle X \quad \textbf{where}\ \ x \gg y = y$$

と書ける．

拡張行列乗算を用いた 3 つの例を以下に示す．

例 1 入力の縦ベクトル Y に対して，あるベクトルの集合の全てのベクトルとの距離の 2 乗を求める問題を考える．二次元配列はベクトルの集合であると見ることができる．ベクトルの集合を表わす二次元配列を X とおく．ここで X の各行は各ベクトルに対応している．この問題は，拡張行列乗算を用いて

$$X \left\langle\!\!\left\langle \begin{matrix} \otimes \\ + \end{matrix} \right\rangle\!\!\right\rangle Y \quad \textbf{where}\ \ x \otimes y = (x-y)^2$$

と書くことができる．

例 2 入力の縦ベクトル Q と同じベクトルが，あるベクトルの集まりの中に含まれているどうか判定する問題を考える．ベクトルの集合を表す二次元配列を X と置く．ここで X の各行が各ベクトルに対応している．この問題は，以下のように拡張行列乗算を用いて記述できる．

$$Z = \mathsf{reduceCol}_{\mathrm{M}}\ (\vee)\ \left(X \left\langle\!\!\left\langle \begin{matrix} = \\ \wedge \end{matrix} \right\rangle\!\!\right\rangle Q \right)$$

ここで，$\vee$ は論理和，$\wedge$ は論理積を表す．拡張行列乗算の中の $=$ は真偽値を返す比較演算子である．まず，最初の拡張行列乗算を用いて縦ベクトル Q と X の各行とが同じかどうか調べる．1 つでも同じものがあればよいので，得られた結果を $\vee$ で縮約する．得らえた値 Z は，1×1 二次元配列であり，その要素 z_{00} が求める結果である．

例 3 n 個の頂点を持つ無向グラフが隣接行列で表現されているときに，全ての最短路を求める計算は，隣接行列を X とすると，n 個の頂点を持つ無効グラフの全ての頂点間の最短路を求める問題を考えよう．i 番目の頂点と j 番目の頂点の重みを x_{ij} とする行列 X を考える．ここで，i 番目の頂点と j 番目の頂点の間に辺が存在しないときは，$x_{ij} = \infty$ とする．すると，この問題は

$$Z^1 = X$$
$$Z^n = X \left\langle\!\!\left\langle \begin{matrix} + \\ \downarrow \end{matrix} \right\rangle\!\!\right\rangle Z^{n-1} \quad \textbf{where}\ \ x \downarrow y = \textbf{if}\ x\ < y\ \textbf{then}\ x\ \textbf{else}\ y$$

となり，Z^n に全ての最短路が求まる．

このとき，$+$ と $\downarrow$ の性質により，拡張行列乗算が結合的になり，さらに同じ値を計算している部分をまとめることで，より効率的に計算できる．

3.3 拡張行列スカラー積

拡張行列スカラー積は以下のように定義される．

$$Z = X \langle\!\langle \otimes \| y \overset{\mathrm{def}}{\Longleftrightarrow} z_{ij} = x_{ij} \otimes y$$

$$Z = x \| \otimes \rangle\!\rangle Y \overset{\mathrm{def}}{\Longleftrightarrow} z_{ij} = x \otimes y_{ij}$$

前者が行列スカラー積，後者がスカラー行列積の演算子を一般化したものである．

$\mathsf{map}_{\mathrm{M}}$ は以下のように

$$\mathsf{map}_{\mathrm{M}}\ f\ X = X \langle\!\langle \otimes \| y \quad \textbf{where}\ \ x \otimes y = f\ x$$

と拡張行列スカラー積を用いて書くことができる．また，逆に拡張行列スカラー積も $\mathsf{map}_{\mathrm{M}}$ を用いて，

$$X \langle\!\langle \otimes \| y = \mathsf{map}_{\mathrm{M}}\ (\otimes y)\ X$$
$$x \| \otimes \rangle\!\rangle Y = \mathsf{map}_{\mathrm{M}}\ (x \otimes)\ Y$$

と記述できる．以後，本論文では，表記が簡潔するため拡張行列スカラー積の表記として $\mathsf{map}_{\mathrm{M}}$ を用いる．

拡張行列スカラー積を用いた 2 つの例を示す．

例 4 実数を表す型を $\mathbb{R}$ とすると，$matrix\ (\mathbb{R}\times\mathbb{R})$ に対して

$$\begin{pmatrix} (a,b) & (c,d) \\ (e,f) & (g,h) \end{pmatrix}$$

を入力とし，

$$\begin{pmatrix} a/b & c/d \\ e/f & g/h \end{pmatrix}$$

を返すような関数を考える．この関数は次のように $\mathsf{map}_{\mathrm{M}}$ で書くことができる．

$$\begin{aligned} &\mathsf{map}_{\mathrm{M}}\ f\ X \\ &\quad \textbf{where} \quad f\ (x,y) = x/y \end{aligned}$$

例 5 二次元配列 X の各行の最小値が何列目に位置するか求める関数を考える．この関数は以下のように定義できる．

$$\begin{aligned} &\mathsf{map}_{\mathrm{M}}\ f\ (\mathsf{reduceRow}_{\mathrm{M}}\ (\oplus)\ (\mathsf{map}_{\mathrm{M}}\ g\ X)) \\ &\quad \textbf{where} \quad g\ x = (x,1,1) \\ &\qquad\qquad (x,i,n)\oplus(y,j,m) \\ &\qquad\qquad\quad = \quad \textbf{if}\ x<y \\ &\qquad\qquad\qquad\quad \textbf{then}\ (x,i,n+m) \\ &\qquad\qquad\qquad\quad \textbf{else}\ (y,j+n,n+m) \\ &\qquad\qquad f\ (x_1,x_2,x_3) = x_2 - 1 \end{aligned}$$

式中の 3 つ組は，最小値，最小値の位置，最小値を含む範囲を表す．n 個の中で i 番目が最小値 x であり，m 個の中で j 番目が最小値 y であったとすると，$n+m$ 個の中での最小値は位置 i の x または，$n+m$ 個の中で $n+i$ 番目の y である．g は，ある値はある値 1 つだけの中で最小値は 1 番目の値であり，その値が最小値であることを表す．3 つ組の中で最終的に求めたいものは最小値の位置のみであるから，f で 2 番目の値を返している．最後に 1 を引いているのは，行列のインデックスが 0 から始まるためである．この $\oplus$ が結合的であることは計算の意味から容易に示せる．

3.4 拡張行列加算

拡張行列加算は，行列演算子での行列加算の演算子を一般化したものであり，結合的な演算子 $\oplus$ を用いて次のように定義する．

$$Z = X\ \langle\!|\oplus|\!\rangle\ Y \overset{\text{def}}{\iff} z_{ij} = x_{ij}\oplus y_{ij}$$

ここで，X，Y，Z は同じサイズであり，型は $matrix\ \mathcal{A}$ である．$\oplus$ の型は $\mathcal{A}\to\mathcal{A}\to\mathcal{A}$ となる．

例 6 二次元配列のリスト

$$\left[\begin{pmatrix} a & b \\ c & d \end{pmatrix}, \begin{pmatrix} x & y \\ z & w \end{pmatrix}\right]$$

をリストの二次元配列

$$\begin{pmatrix} [a,x] & [b,y] \\ [c,z] & [d,w] \end{pmatrix}$$

に変換する問題を考える．このような，$list\ (matrix\ \mathcal{A})$ である $[X_1,\ldots,X_n]$ から，$matrix\ (list\ \mathcal{A})$ への変換は次の式で与えることができる．

$$\mathsf{foldr}_{\mathrm{L}}\ (\langle\!|{+\!\!+}|\!\rangle)\ NIL\ (\mathsf{map}_{\mathrm{L}}\ (\mathsf{map}_{\mathrm{M}}\ ([\cdot]))\ [X_1,\ldots X_n])$$

ここで NIL は $nil_{ij} = []$ である．

拡張行列加算は二次元配列上の並列スケルトン $\mathsf{zipWith}_{\mathrm{M}}$ を用いて容易に書くことができる．加えて，拡張行列加算と拡張スカラー行列積を用いて $\mathsf{zipWith}_{\mathrm{M}}$ を

$$\begin{aligned} &\textbf{data}\ \mathcal{F} = FX\ \mathcal{X} \mid FY\ \mathcal{Y} \\ &\mathsf{zipWith}_{\mathrm{M}}\ f\ X\ Y = \\ &\quad \mathsf{map}_{\mathrm{M}}\ f'\ (X'\ \langle\!|{+\!\!+}|\!\rangle\ Y') \\ &\qquad \textbf{where} \quad f'\ [FX\ a, FY\ b] = f\ a\ b \\ &\qquad\qquad\quad X' = \mathsf{map}_{\mathrm{M}}\ ([\cdot])\ X^{\dagger} \\ &\qquad\qquad\quad Y' = \mathsf{map}_{\mathrm{M}}\ ([\cdot])\ Y^{\dagger} \\ &\qquad\qquad\quad X^{\dagger} = \mathsf{map}_{\mathrm{M}}\ FX\ X \\ &\qquad\qquad\quad Y^{\dagger} = \mathsf{map}_{\mathrm{M}}\ FY\ Y \end{aligned}$$

と記述することができる．但し $\mathcal{X}$ は X の型であり $\mathcal{Y}$ は Y の型である．ここで，**data** はデータ型定義である．上では，データ構成子が FX と FY である新な型 $\mathcal{F}$ を宣言している．本論文では，結合的な演算子および関数を用いるときには，$\langle\!|\oplus|\!\rangle$ を用い，そうでない場合に $\mathsf{zipWith}_{\mathrm{M}}$ を用いる．

4. 拡張行列演算によるデータマイニングアルゴリズムの記述

本節では，いくつかのデータマイニングのアルゴリズムを記述することにより，拡張行列演算を用いて簡潔にアルゴリズム記述ができることを示す．

4.1 Boolean Search

表 1 に示されるスーパーマーケットの取引記録 M を考える．ここで “ある品物を買った” ということを原子論理式とする論理式を与え，それぞれの人について，その人の買ったものが論理式を満たす場合に $true$，そうでない場合 $false$ を返す問題を考える．この問題を本論文では Boolean Search と呼ぶ．たとえば，表中の A，B，C の中で，“ミカン” という入力に対し，A と B が $true$ になり，C が $false$ になる．そして，

表 1 スーパーマーケットの取引記録
Table 1 Transaction record in a supermarket

名前	ミカン	リンゴ	バナナ	···	メロン
A	3	0	4	···	0
B	12	2	2	···	1
C	0	1	0	···	2
⋮	⋮	⋮	⋮	⋱	⋮
ZZZZ	0	3	5	···	0

“ミカン $\wedge \neg$バナナ” という入力に対して，A，B，C ともに *false* となる．これはミカンを購入し，なおかつバナナを購入していないものが，3 人の中にいないためである．

入力の論理式として，積和標準型の論理式

$$Q_1 \vee Q_2 \vee \cdots \vee Q_n$$

を考える．ここで，全ての i に対し Q_i は $\vee$ を含まない．この Q_i は，その品物が式にどの形で含まれているかによって，縦ベクトルへ対応させることができ，これを Y_i とおく．$Q_1 =$ ミカン，$Q_2 = \neg$バナナ，$Q_3 =$ ミカン $\wedge \neg$バナナ の場合の例を**表 2** に示す．このとき Y_3 は

$$Y_3 = (\text{肯定 含まない 否定 } \cdots \text{ 含まない})^T$$

となる．

拡張行列演算を利用して，Boolean Search のアルゴリズムを**図 2** のように記述できる．以降，このアルゴリズムを BS1 と書く．図 2 での Z が求めるべき結果である．

拡張行列演算を用いて，アルゴリズムを記述する方法は一意でない場合がある．たとえば，Boolean Search のアルゴリズムは，**図 3** のようにも記述できる．以降，このアルゴリズムを BS2 と書く．BS1 と BS2 との違いは第 6 節での実験にて示す．

4.2 k-means アルゴリズム

k-means アルゴリズム[12] はクラスタリングアルゴリズムの 1 つであり，クラスタの数 k を予め定めておき，n 個の点を k 個のクラスタに分割する．k-means アルゴリズムは以下のステップで計算される．

ステップ 1 k 個の代表点を n 個の点の中からランダムに選ぶ．1 個の代表点に 1 個のクラスタが対応する．

ステップ 2 n 個の点をもっとも近い代表点の属するクラスタに割り付ける．

ステップ 3 得られた k 個のクラスタに関して，あらたな k 個の代表点を計算しなおす．

ステップ 4 ある評価関数が減少しなくなるまで，ステップ 2，3 を繰り返す．

評価関数として，クラスタ内でのクラスタの代表点との距離の 2 乗和のクラスタ間総和を用いる．ここでは，距離としてユークリッド距離を考える．k-means アルゴリズムは貪欲であり，結果は，初期の k 個の点へ大きく依存する極大値に収束する．そのため実際には，最大値を得るために初期の k 個の点を適切に変更しつつ，上記のアルゴリズムを反復する必要がある．なお，今回はその部分は本質的でないため考慮しない．

ここで，

$$S = (1\ 2\ \cdots\ k)$$

という $1 \times k$ 二次元配列を定義する．各点を表す横ベクトルを n 個縦に並べたものを X とし，K を k 個の代表点を横にならべたものであるとする．すると，上のステップ 2，3，4 は**図 4** のような拡張行列演算を用いた再帰関数 *kmeans_step234* にて表現できる．

図 4 中の C の各要素は，最も近いの代表点への距離，最も近い代表点の番号，k の 3 つ組である．*MSE* は 1×1 二次元配列であり，mse_{00} は評価関数の値を表している．$\odot$ は，第 3 節の例 4 に同様なものが示されている．表中の

$$C \left\langle\!\!\left\langle \begin{matrix} \otimes' \\ \oplus' \end{matrix} \right\rangle\!\!\right\rangle S$$

は

$$k = 3, C = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 1 \\ 3 \end{pmatrix}$$

のとき，i 行目の c_{i0} 要素のみ 1 であるような $n \times k$ 行列

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

を返す．

4.3 連 想 計 算

ここでは，DualNAVI[19] や GETA[9] で扱う連想計算がこの枠組みで扱えることを示す．連想計算は、ある与えられた文書からそれに類似した文書群を返すというように，与えられた要素に類似する要素群を計算する。通常 GETA などの連想計算では，もっとも類似している n 個の要素を返すのであるが，これは容易に $\mathsf{reduceCol}_{\mathrm{M}}$ を用いて達成できるため，ここでは与えられた文書と他の文書との類似度を計算する過程ま

表 2 論理式と二次元配列との対応
Table 2 Representing logic formula with two-dimensinal array

論理式	ミカン	リンゴ	バナナ	...	メロン
ミカン	肯定	含まない	含まない	...	含まない
¬バナナ	含まない	含まない	否定	...	含まない
ミカン ∧ ¬バナナ	肯定	含まない	否定	...	含まない

$$
\begin{array}{lll}
A = \mathsf{map}_{\mathrm{L}}\ \left(M \left\langle\!\!\left\langle \begin{smallmatrix}\otimes\\ \oplus\end{smallmatrix} \right\rangle\!\!\right\rangle\right)\ [Y_1, Y_2, \ldots, Y_n] & \{\wedge, \neg \text{ を先に処理する }\} \\
Z = \mathsf{foldr}_{\mathrm{L}}\ (\langle\!|\vee|\!\rangle)\ \mathit{FAL}\ A & \{\text{ 最後に } \vee \text{ を計算する }\} \\
\quad \textbf{where} \quad x \otimes y = & \textbf{if } y = \text{否定} \wedge x \neq 0 \textbf{ then} \quad \mathit{false} \\
& \textbf{else if } y = \text{肯定} \wedge x = 0 \textbf{ then} \quad \mathit{false} \\
& \textbf{else} \quad \mathit{true} \\
x \oplus y = & x \wedge y \\
\mathit{fal}_{ij} = & \mathit{false} \quad \{\vee \text{ の単位元 }\}
\end{array}
$$

図 2 Boolean Search アルゴリズム BS1
Fig. 2 Boolean Search algorithm BS1

$$
\begin{array}{lll}
\{\text{ まず } \mathit{matrix\ list} \text{ を } \mathit{list\ matrix} \text{ に変換しておく }\} & & \\
Y = \mathsf{foldr}_{\mathrm{L}}\ (\langle\!|{+\!\!+}|\!\rangle)\ \mathit{NIL}\ (\mathsf{map}_{\mathrm{L}}\ (\mathsf{map}_{\mathrm{M}}\ ([\cdot]))\ [Y_1, Y_2, \ldots, Y_n]) & & \\
Z = \mathsf{map}_{\mathrm{M}}\ f\ \left(M \left\langle\!\!\left\langle \begin{smallmatrix}\otimes'\\ \oplus'\end{smallmatrix} \right\rangle\!\!\right\rangle Y\right) & & \\
\quad \textbf{where} \quad x \otimes y = & \textbf{if } y = \text{否定} \wedge x \neq 0 \textbf{ then} \quad \mathit{false} & \\
& \textbf{else if } y = \text{肯定} \wedge x = 0 \textbf{ then} \quad \mathit{false} & \\
& \textbf{else} \quad \mathit{true} & \{\text{BS1 と同じ }\} \\
x \oplus y = & x \wedge y & \{\text{BS1 と同じ }\} \\
x \otimes' y = & \mathsf{map}_{\mathrm{L}}\ (x\otimes)\ y & \{\text{ 構造の変化に対応 }\} \\
x \oplus' y = & \mathsf{zipWith}_{\mathrm{L}}\ (\oplus)\ x\ y & \{\text{ 構造の変化に対応 }\} \\
f\ x = & \mathsf{foldr}_{\mathrm{L}}\ (\vee)\ \mathit{false}\ x & \\
\mathit{nil}_{ij} = & [] &
\end{array}
$$

図 3 Boolean Search アルゴリズム BS2
Fig. 3 Boolean Search algorithm BS2

でを考える．

4.3.1 文書間の類似度の計算

文書と文書の類似度を計算するためには，何らかの尺度を用いて類似度を定義する必要がある．ここでは，IDF（Inverse Document Frequecy）といわれる尺度[18] を用いる．この尺度では，頻出語の類似度への影響が小さい．たとえば，この尺度の下では「僕」や「私」などの単語はあまり類似度に寄与しない．

このとき類似度の計算は以下のようになる．

$$sim(d_1|d_2) = \frac{\sum_t tf(t|d_1) \times tf(t|d_2) \times idf(t)}{norm(d_1)}$$

$tf(t|d)$ は文書 d 中の単語 t の出現頻度を表す．また

$$norm(d) = \sum_t tf(t|d)$$

であり，

$$idf(t) = \log \frac{N}{df(t)}$$

である．N は総文書数であり，$df(t)$ は単語 t を含む文書数である．

4.3.2 拡張行列演算での表現

行に文書，列に単語を対応させた行列 X を考える．$x_{ij} = tf(t_j|d_i)$ であり，X の各要素が文書中の単語の出現頻度を表している．

文書数 N を求めるのは容易なので，N は既知であるとする．また，入力 Q は重みつき文書集合である．たとえば，第 0 番目の文書に類似しているものを求める場合，入力 Q は q_{00} のみが 1 であり，他の要素が 0 である．そして，第 0 番目と第 1 番目の文書両方につ

$$
\begin{array}{l}
kmeans\_step234\ K\ oldmse = \\
\quad \{\ \textbf{ステップ 2:}\} \\
\quad\quad D = X \left\langle\!\!\left\langle \begin{matrix}\otimes \\ + \end{matrix} \right\rangle\!\!\right\rangle K \\
\quad\quad\quad \textbf{where} \quad x \otimes y = (x-y)^2 \\
\quad \{\ \textbf{ステップ 3:}\} \\
\quad\quad C = \mathsf{reduceRow}_{\mathrm{M}}\ (\odot)\ (\mathsf{map}_{\mathrm{M}}\ f\ D) \\
\quad\quad MSE = \mathsf{reduceCol}_{\mathrm{M}}\ (+)\ (\mathsf{map}_{\mathrm{M}}\ \pi_1\ C) \\
\quad\quad\quad \textbf{where} \quad f\ x = (x,1,1) \\
\quad\quad\quad\quad\quad (x,i,n) \odot (y,j,m) = \textbf{if}\ x < y\ \textbf{then}\ (x,i,n+m)\ \textbf{else}\ (y,j+n,n+m) \\
\quad\quad\quad\quad\quad \pi_1\ (x,i,k) = x \\
\quad \{\ \textbf{ステップ 4:}\} \\
\quad\quad \textbf{if}\ mse_{00} \geq oldmse\ \textbf{then} \\
\quad\quad\quad \mathsf{map}_{\mathrm{M}}\ (\pi_2)\ C \qquad \{\ \text{それぞれの点が最も近い代表点の番号を返す}\ \} \\
\quad\quad \textbf{else} \\
\quad\quad\quad kmeans\_step234\ \left(X^T \left\langle\!\!\left\langle \begin{matrix}\times \\ + \end{matrix} \right\rangle\!\!\right\rangle \left(\mathsf{map}_{\mathrm{M}}\ \pi_2\ C \left\langle\!\!\left\langle \begin{matrix}\otimes' \\ \oplus' \end{matrix} \right\rangle\!\!\right\rangle S \right) \right)\ mse_{00} \\
\quad\quad\quad\quad \textbf{where} \quad \pi_2\ (a,b,c) = b \\
\quad\quad\quad\quad\quad\quad x \otimes' y = \textbf{if}\ x = y\ \textbf{then}\ 1\ \textbf{else}\ 0 \\
\quad\quad\quad\quad\quad\quad x \oplus' y = x\ \{\ \text{この演算子は計算に使用されない}\ \}
\end{array}
$$

図 4 k-means アルゴリズム: ステップ 2, 3, 4
Fig. 4 k-means algorithm: steps 2,3,4

いて似ているものを考えている場合には，入力 Q は $q_{00} = q_{10} = 1$ でその他が 0 である．

このとき，**図 5** のように拡張行列演算を行うことで，入力の文書集合に対するそれぞれの文書の類似度 Z が求められる．

5. 拡張行列演算を用いた並列分散化

本節では，拡張行列演算の並列分散化について議論する．

並列分散ではデータをどのように分割するのか，そして，分割したものをどのように扱うのかを考えなければならない．ここまでは，定式化を行うために行列の表現方法に依存しない議論をしてきたが，ここでは実際にこの枠組みを実装することを考え，具体的な並列分散化手法について述べる．

5.1 二次元配列のブロック分割

二次元配列の配列を格子状に分割するブロック分割（**図??**）は一方向のみの分割と比べ，並列分散計算と相性がよい．

まず，疎な二次元配列を扱う場合を考える．ここでは，疎な二次元配列の表現法として圧縮列格納[2] または圧縮行格納[2] を用いる場合を考える．一方向のみの分割だと，その表現と分割の方向によって，使用メモリ量のオーダが列数または行数より小さくできない場合がある．たとえば，圧縮列格納の場合，行方向に分割すると二次元配列の幅のオーダよりはサイズを小さくすることができない．また，二次元配列間の演算の計算量が，お互いの格納の仕方により変わることもある．その場合，一方向のみの分割しか許さないと，時間と空間の極端なトレードオフに遭遇してしまう場合がある．実際に GETA はこの問題に遭遇した．ブロック分割はそこに中間的な選択肢を与えることが，松田らによって示されている[23]．

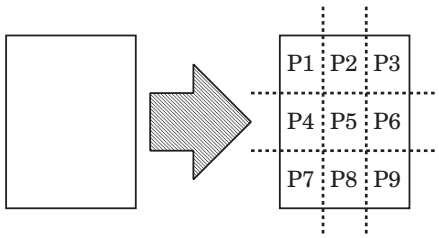


図 6 ブロック分割
Fig. 6 Blockwise partitioning

次に，プロセッサ間の通信について考える．拡張行列演算では拡張行列乗算と残りの 2 つとを合わせて用いる場合が多く，拡張行列加算と拡張行列スカラー積

$$
\begin{aligned}
&Y_1 = X^T \left\langle\!\!\left\langle \begin{matrix}\times\\ +\end{matrix} \right\rangle\!\!\right\rangle Q \quad \{\text{ 入力文書に含まれる単語ごとの頻度 }\}\\
&Y_2 = X^T \left\langle\!\!\left\langle \begin{matrix}\otimes\\ +\end{matrix} \right\rangle\!\!\right\rangle Q \quad \{\text{ 単語ごとの } df\}\\
&Y = \mathsf{zipWith}_{\mathrm{M}}\ pair\ Y_1\ Y_2\\
&Z = \mathsf{map}_{\mathrm{M}}\ f\ \left(X \left\langle\!\!\left\langle \begin{matrix}\otimes'\\ \oplus\end{matrix} \right\rangle\!\!\right\rangle Y\right)\\
&\quad \textbf{where} \quad x \otimes y = \textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } 1\\
&\qquad\qquad pair\ x\ y = (x, y)\\
&\qquad\qquad x \otimes' (w, df) = \textbf{if } x = 0 \textbf{ then } (0,0) \textbf{ else } (x \times w \times \log(N/df), x)\\
&\qquad\qquad f\ (x_1, x_2) = x_1/x_2
\end{aligned}
$$

図 5 連想計算
Fig. 5 Association computation

のみでは表現力が不足する場合が多い．一方向の分割では拡張行列乗算の際に互いの分割方向が異なっていなければならず，通信によって適宜方向を変えてやる必要がある．さらに，分割方向が異なっている場合には拡張行列加算においても通信が発生してしまう．それに対し，ブロック分割では p^2 個のプロセッサを用いて縦に p 個，横に p 個に分割していれば，拡張行列乗算のときに分割方法を気にする必要がなくなり，拡張行列加算および拡張行列スカラー積の場合も通信が発生しない．

最後に，ブロック分割を用いることで効率のよいアルゴリズムを適用することができることを述べる．たとえば，拡張行列乗算の並列アルゴリズムの記述は，ブロック分割でなくとも可能である．しかし，ブロック分割する Cannon のアルゴリズム[10] を拡張行列乗算として用いると，素朴な一方向のみの分割の上での拡張行列乗算と比べて，通信コストのオーダを下げることができる[10]．

5.2 演算子 $\oplus$ の結合性

ブロック分割において，拡張行列乗算は簡単には並列化できない．もし拡張行列乗算で用いられる演算子 $\oplus$ に対し結合則と交換則が成り立たなければ，一般には，Cannon のアルゴリズムは分割する前と違った値を返してしまう．本研究では，Cannon のアルゴリズムに簡単な変形を行うことにより，演算子 $\oplus$ の可換性を必要としないアルゴリズムに変えたものを用いている．そのため，演算子 $\oplus$ に結合性という制約を加えることで，効率がよく性質のよいブロック分割を用いることができる．

結合的な演算子を求めるのは一般に簡単ではない．しかし，既に結合的な演算子を再帰的な関数定義から系統的に求める手法[6] も提案されていて，さらに自動化手法[21] も研究されているため，結合的演算子が必要であるというこの枠組みの制約は表現力をあまり制限しない．

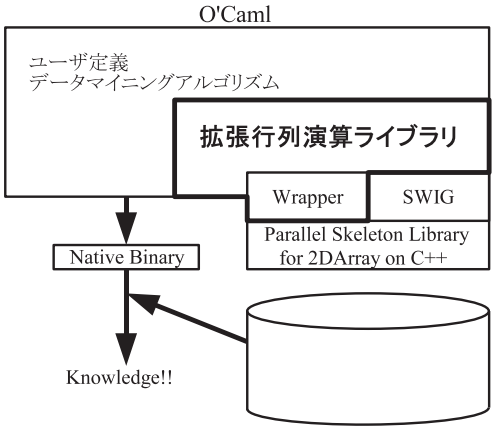


図 7 環境の概観
Fig. 7 Overview of our environment

6. 実　　装

6.1 概　　観

我々は拡張行列演算を**図 6** のような環境として実現した．拡張行列演算を利用するための環境を関数型言語 O'Caml[16] のライブラリとして構築した．ユーザは，**図 7** のように拡張行列演算を使ったプログラムを記述することができる．これは，BS1（図 2）のアルゴリズムを我々の環境の上でプログラムしたものであり，アルゴリズムを自然にプログラムで記述できている．図 7 中の `Dmatrix` は，分散された二次元配列を扱うモジュールである．拡張行列乗算

$$X \left\langle\!\!\left\langle \begin{matrix}\otimes\\ \oplus\end{matrix} \right\rangle\!\!\right\rangle Y$$

```
type bqelem = Contains | ContainsAsNot | DontCare
let boolean_search1 transaction_matrix query_matrices =
  let tm     = transaction_matrix in
  let fl     = Dmatrix.gen (tm#rows()) 1 (tm#divs()) false in
  let otimes x y = if      y = Contains      && x != 0 then false
                   else if y = ContainsAsNot && x == 0 then false
                   else                                     true  in
  let oplus x y   = x && y  in
  let anss   = List.map (Dmatrix.mul otimes oplus tm) query_matrices in
    List.fold_right (Dmatrix.zip (fun x y -> x || y)) anss fl
```

図 8 我々の環境でのプログラミング例
Fig. 8 A program for BS1 on our environment

は，`Dmatrix.mul` $(\otimes)$ $(\oplus)$ `X Y` というプログラムに対応する．ここで `Dmatrix.mul` は $(\mathcal{A} \to \mathcal{B} \to \mathcal{C}) \to (\mathcal{C} \to \mathcal{C} \to \mathcal{C}) \to \mathit{matrix}\ \mathcal{A} \to \mathit{matrix}\ \mathcal{B} \to \mathit{matirx}\ \mathcal{C}$ という型を持つ高階関数である．同様に，拡張行列スカラー積 $\mathsf{map}_{\mathrm{M}}$ は `Dmatrix.map` に，拡張行列加算 $\langle\!|\oplus|\!\rangle$ および $\mathsf{zipWith}_{\mathrm{M}}$ は `Dmatrix.zip` に対応している．モジュール `Dmatrix` を実現するのに，江本らによる C++と MPI で実装されたブロック分割を行う二次元配列に対する並列スケルトンの実装[22] を使用した．そして C++オブジェクトを O'Caml から利用するのに SWIG を用いた．

このようにして記述されたプログラムは，O'Caml コンパイラおよび C++コンパイラにより，MPI 上の実行ファイルに変換される．

6.2 実　　験

この枠組みであるアルゴリズムの記述方法が複数存在する場合に，どのように記述すれば実行速度の点で効率がよいのか調べるため，第 4.1 節で定義した 2 つの Boolean Search アルゴリズムについて実行速度を比較した．ここで BS1 および BS2 において，積和標準形での $\vee$ の出現数 n を 7 と設定した．

実験には LAN で接続された PC クラスタを用いた．各マシンの環境は以下の通りである．

- ハードウェアおよび OS
  - Pentium 4 Xeon 2.0-GHz 2CPU SMP
  - 1GB のメモリ
  - ギガビットイーサネット
  - FreeBSD 4.10
- コンパイラおよびライブラリ
  - gcc 2.95
  - MPICH 1.2.2
  - O'Caml 3.08.1
  - SWIG 1.3.22

このとき得られた結果は，**表 3**，**表 4** に示される通りである．表中の 400，800 などは取引記録を表す二次元配列のサイズが 400×400，800×800 などであることを表す．また，表中の要素は経過時間を秒で表したものである．

アルゴリズムの並列性について検証する．サイズ 400×400，使用プロセッサ数 1 の場合は 2 つのアルゴリズムの実行速度にあまり差はない．このことからサイズ 400×400 の場合の計算のコストはどちらのアルゴリズムもほぼ等しいと考えられる．しかし，サイズ 1200×1200，使用プロセッサ数 9 の場合には，各プロセッサあたりの二次元配列のサイズが 400×400 になるにもかかわらず BS2 の方が速い．Cannon のアルゴリズムは二次元配列とベクトルの積を計算する場合，通信コストと計算コストは同程度になる[10]．よって，BS1 の方が通信コストが大きいといえる．通信のコストが大きいプログラムは並列化の際に遅くなる場合がある．サイズ 400×400 の場合の台数効果のプロット**図 8**，**図 9** を見てみると，1 回しか拡張行列演算を行わない BS2 では台数効果がでているが，通信コストの大きい BS1 はプロセッサ数 1 の場合と比較して遅くなっている．

実験より，我々の環境において，ベクトルのリストと行列に対し同じ演算子を用いた拡張行列乗算を行う場合は，拡張行列乗算の回数の少くしたプログラムの方が速く並列実行されると言える．

7. 拡張行列演算上のプログラム変換

拡張行列演算では，計算パターンが拡張行列乗算，拡張行列加算，拡張行列スカラー積の 3 つと少ない．プログラムは演算子が適切に定義された拡張行列演算を組み合わせることで記述されているため，その組み合わせ方を代数的に取り扱うことにより，プログラム

$$\mathsf{foldr}_{\mathrm{L}}\ (\langle\!|\odot|\!\rangle)\ U\ \left(\mathsf{map}_{\mathrm{L}}\ \left(X \left\langle\!\!\left\langle \begin{matrix}\otimes\\ \oplus\end{matrix} \right\rangle\!\!\right\rangle\right)\ [Y_1, Y_2, \ldots, Y_n]\right)$$
$$\textbf{where}\quad u_{ij} = e$$

$$\Updownarrow$$

$$\mathsf{map}_{\mathrm{M}}\ f\ \left(X \left\langle\!\!\left\langle \begin{matrix}\otimes'\\ \oplus'\end{matrix} \right\rangle\!\!\right\rangle Y\right)$$
$$\begin{aligned}\textbf{where}\quad & Y = \mathsf{foldr}_{\mathrm{L}}\ (\langle\!|+\!+|\!\rangle)\ \mathit{NIL}\ (\mathsf{map}_{\mathrm{L}}\ (\mathsf{map}_{\mathrm{M}}\ ([\cdot]))\ [Y_1, \ldots, Y_n]\\ & x \oplus' y = \mathsf{map}_{\mathrm{L}}\ (x\otimes)\ y\\ & x \otimes' y = \mathsf{zipWith}_{\mathrm{L}}\ (\oplus)\ x\ y\\ & f\ x = \mathsf{foldr}_{\mathrm{L}}\ (\odot)\ e\ x\\ & nil_{ij} = []\end{aligned}$$

図 11 プログラム変換例
Fig. 11 Example for promotion

$$Z = \mathsf{zipWith}_{\mathrm{M}}\ (/)\ D\ \left(X^{\dagger T} \left\langle\!\!\left\langle \begin{matrix}\times\\ +\end{matrix} \right\rangle\!\!\right\rangle \left(X^{\dagger} \left\langle\!\!\left\langle \begin{matrix}\times\\ +\end{matrix} \right\rangle\!\!\right\rangle Q\right)\right)$$

図 12 連想計算'
Fig. 12 Association Computation'

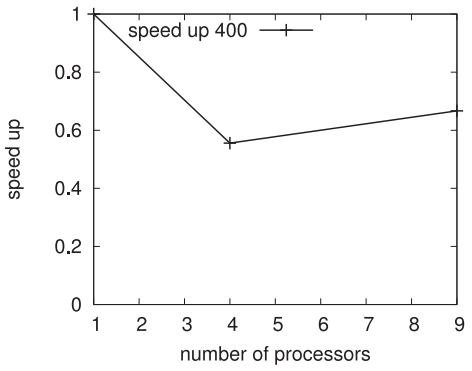


図 9 台数効果: BS1（サイズ 400 × 400）
Fig. 9 Speed up : BS1 (size:400 × 400)

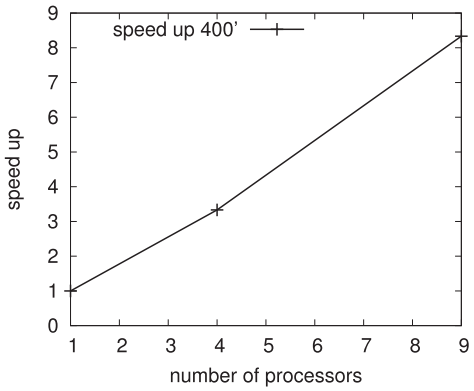


図 10 台数効果: BS2（サイズ 400 × 400）
Fig. 10 Speed up : BS2 (size:400 × 400)

表 3 実行時間:BS1
Table 3 Elapsed time: BS1

	データサイズ $N \times N$			
プロセッサ数	400	800	1200	1600
1	1.0	4.1	10	15
4	1.8	7.9	17	31
9	1.5	6.9	15	26

表 4 実行時間:BS2
Table 4 Elapsed time: BS2

	データサイズ $N \times N$			
プロセッサ数	400	800	1200	1600
1	1.0	8.5	35	100
4	0.30	1.9	6.4	17
9	0.12	0.72	2.0	4.9

変換を行うことができる．拡張行列演算では，計算パターンが少ないため，その組み合わせを限定しやすく，プログラム変換を適用しやすい．

プログラム変換により，拡張行列演算を用いて記述されたプログラムの効率化の指針を与えることができる．そこで本節では，この枠組みの上で効率的なプログラムを書くためにはどのようなプログラム変換が用いられるべきであるかを簡単に紹介する．

第 4.1 節での BS2 は，素朴な定義である BS1 を，**図 10** のようなプログラム変換をすることで得ることができる．実験で示されたように，このプログラム変換を行うことで，素朴に定義されたプログラムを効率

よい並列分散プログラムに変換することができる．この枠組みで通信量はボトルネックになる場合があり，その場合このような変換が有用である．

また，計算量および通信量を減らすために前処理をすることが考えられる．たとえば，第 4.3 節での，尺度に IDF を用いた連想計算を考える．第 4.3 節では，図 5 のように素朴に何の前処理も行わず書き下した．しかし，IDF を用いた類似度の計算は，通常は可能かぎり前処理をしたのちに計算される．$norm(d)$ や $idf(t)$ を予め計算しておき $X^\dagger$，D を

$$x^\dagger_{ij} = x_{ij}\sqrt{idf(t_j)}$$

および

$$d_{i0} = norm(d_i)$$

と定義すると，**図 11** のようなプログラムが得られる．ここでは，図 5 に表れたような log などの重い関数がなくなり，基本的な演算子 $\times, +$ で記述されているので演算子に相当する関数の呼び出す必要がなくなるため高速に計算できる．また，明らかにこのとき拡張行列演算は結合的になるため，

$$X^{\dagger T} \left\langle\!\!\left\langle \begin{matrix}\times\\ +\end{matrix} \right\rangle\!\!\right\rangle X^\dagger$$

を前計算しておくことが可能である．しかし，このように可能なら前処理をしておくというプログラム変換は，拡張行列演算の上ではまだあまり議論されていない．

この枠組みで記述されるプログラムは

$$X \left\langle\!\!\left\langle \begin{matrix}\otimes'\\ \oplus'\end{matrix} \right\rangle\!\!\right\rangle \left(Y \left\langle\!\!\left\langle \begin{matrix}\otimes\\ \oplus\end{matrix} \right\rangle\!\!\right\rangle Z\right)$$

という形をしていることが多く，Z が入力であり，X と Y は変化しない場合が多い．このとき

$$\left(X \left\langle\!\!\left\langle \begin{matrix}\otimes'\\ \oplus'\end{matrix} \right\rangle\!\!\right\rangle Y\right) \left\langle\!\!\left\langle \begin{matrix}\otimes\\ \oplus\end{matrix} \right\rangle\!\!\right\rangle Z$$

と変形できると

$$X \left\langle\!\!\left\langle \begin{matrix}\otimes'\\ \oplus'\end{matrix} \right\rangle\!\!\right\rangle Y$$

を前処理しておくことで，効率のよいプログラムに変えることができる場合がある．実際には以下の条件

$$\begin{aligned}
&a \otimes' (\textstyle\bigoplus_i b_i) = \bigoplus_i a \oplus' b_i && \{\text{ 左分配則 }\}\\
&a \otimes' (b \otimes c) = (a \otimes' b) \otimes c\\
&\textstyle\bigoplus_i{}' \bigoplus_j b_{ij} = \bigoplus_j \bigoplus_i{}' b_{ij} && \{\text{ アバイド}^{3)}\}\\
&(\textstyle\bigoplus_i{}' a_i) \otimes b = \bigoplus_i{}' a_i \otimes b && \{\text{ 右分配則 }\}
\end{aligned}$$

が成り立てば必ず前処理できる．なぜなら，

$$\begin{aligned}
&X \left\langle\!\!\left\langle \begin{matrix}\otimes'\\ \oplus'\end{matrix} \right\rangle\!\!\right\rangle \left(Y \left\langle\!\!\left\langle \begin{matrix}\otimes\\ \oplus\end{matrix} \right\rangle\!\!\right\rangle Z\right)\\
&= \textstyle\bigoplus_j{}' x_{ij} \otimes' \left(\bigoplus_k y_{jk} \otimes z_{kl}\right)\\
&= \{\otimes' \text{ の } \oplus \text{ に対する左分配的 }\}\\
&\quad \textstyle\bigoplus_j{}' \left(\bigoplus_k x_{ij} \otimes' (y_{jk} \otimes z_{kl})\right)\\
&= \{\mathrm{a} \otimes' (b \otimes c) = (a \otimes' b) \otimes c\}\\
&\quad \textstyle\bigoplus_j{}' \left(\bigoplus_k (x_{ij} \otimes' y_{jk}) \otimes z_{kl}\right)\\
&= \{\oplus' \text{ と } \oplus \text{ がアバイド }\}\\
&\quad \textstyle\bigoplus_k \left(\bigoplus_j{}' (x_{ij} \otimes' y_{jk}) \otimes z_{kl}\right)\\
&= \{\otimes \text{ が } \oplus \text{ に対して右分配的 }\}\\
&\quad \textstyle\bigoplus_k \left(\bigoplus_j{}' x_{ij} \otimes' y_{jk}\right) \otimes z_{kl}\\
&= \left(X \left\langle\!\!\left\langle \begin{matrix}\otimes'\\ \oplus'\end{matrix} \right\rangle\!\!\right\rangle Y\right) \left\langle\!\!\left\langle \begin{matrix}\otimes\\ \oplus\end{matrix} \right\rangle\!\!\right\rangle Z
\end{aligned}$$

と変換できるためである．

8. 関 連 研 究

本研究に関連の高い研究を紹介する．

我々の枠組みは GETA の計算方法を抽象化したものとも考えられる．そのため GETA で用いる計算の主なものは，この系により記述できる．GETA は ASCII24☆および新書マップ☆☆などで実際に使用されている．また，松田ら[23] は GETA における第 6 節で示したような，行列の分割方向の問題についての解決を葉に木を持つ木構造を利用した通信を用いて解決しようとした．しかし，その場合には 15 個もの関数を適切に定義する必要があった．それに対し，この枠組みでは定義する演算子自体は非常に少数で済む．その一方で 23) では，ある確率以上で正しい結果を返す確率付き連想計算について，集約のプロセスを予め定めておくことにより解決しているのに対し，この枠組みで確率付き連想計算を効率よく表現するのは難しい．

Li らは，データ並列な Java の方言を作成し，それを自分たちの作ったミドルウェアのためのコードに変換するコンパイラを作成した[14]．その環境では，データ並列なループから自動的に，大域的な縮約関数を抽

☆ http://ascii24.com/
☆☆ http://shinshomap.info/

出することができる．このアプローチはデータに対する走査に着目している点では，我々のアプローチに近い．だが，彼らの着目する走査は縮約と反復であり，我々の着目する走査は行列の演算を拡張したものであるという点で大きく異る．また，我々が二次元的なデータに特化した議論をしているのに対し，彼らのシステムでは多次元的なデータも扱える．

Bird は構成的に二次元配列を扱うために，アバイドという条件を導入した[3]．Bird の構成的な二次元配列はあらゆる場所で分割することができる．しかし，その一方その基本的走査を行う際には用いられる演算子がアバイドを満さなければならない．我々の枠組みでは，基本的な走査においては演算子の結合性のみが必要になるが，最適化の条件としてアバイドが出てくる．

9. ま と め

9.1 結　　論

本研究では，データマイニングにおける並列アルゴリズムの記述を簡単にするため，二次元のデータ構造の走査の仕方に着目して拡張行列演算という枠組みを提案し，かつ実装を示した．この枠組みの上の基本的な走査は，行列演算とアナロジーで与えられるため，ユーザは走査を直観的に利用することができる．また拡張行列演算，基本的な計算パターンが少なく簡潔であり，並列分散化などの議論が行いやすく，十分に強力であり，実際に使われているアプリケーションの主な計算をはじめ，様々なアルゴリズムを記述することができる．また，プログラム変換をこの枠組みの上で記述されたプログラムに適用することで，効率のよいプログラムが得られることを実験にて示した．

9.2 今後の課題

表 4 を見ると，今回の実装において並列効果はあらわれているものの，逐次の場合の行列のサイズに対する計算時間が理論的な計算量 $O(n^2)$ より悪くなっている．これは今回の実装における C++ と O'Caml の間の部分でのメモリ管理が原因であると考えられる．拡張行列演算の実用的な実装を行うためには，この原因を詳しく調べることが求められる．

もし，拡張行列演算の上でのプログラム変換規則を発見，整理することができれば，ユーザに効率のようプログラムを記述する指針を与えることができるようになる．特に，前処理することができれば，拡張行列演算を行う回数が減り通信回数や使用メモリなどが減る場合がある．このような変換規則の発見，整理する研究が望まれる．

現実のデータマイニングの問題は対象とするデータが疎になる場合が多い．演算子を拡張した上で，疎な二次元配列を扱うための議論を進めていけば，大規模な対処可能なアルゴリズムの導出や，プログラム変換によりまったく新しい効率的なアルゴリズムを発見できる可能性がある．

データベースとの連携を考えた場合，関係データベースの積にあたる演算をこの枠組みで表現するのは難しい．さらに，この枠組み自体の表現力がどれほどのものであるか，まだよくわかっていない．この枠組みの構成要素を拡張すべきか，拡張するとすれば何をすべきかの議論も今後の課題である．

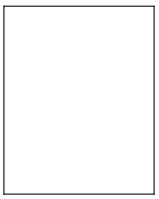
謝辞　本論文を作成するにあたって，定式化手法および記法などに関して様々なアドバイスをいただいた国立情報学研究所の高野明彦氏および東京大学大学院情報理工学系研究科の西岡真吾氏に深く感謝します．

参 考 文 献

1) Banger, C.: Arrays with Categorical Type Constructors, *ATABLE'92 Proceedings of a Workshop and Arrays*, pp. 105–121 (1992).
2) Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C. and der Vorst, H. V.: *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*, SIAM, Philadelphia, PA (1994).
3) Bird, R. S.: Lectures on Constructive Functional Programming, Technical Report Technical Monograph PRG-69, Oxford University Computing Laboratory (1988).
4) Bird, R. S.: *Introduction to Functional Programming Using Haskell*, Prentice Hall (1998). P HAS 98:1 1.Ex.
5) Bird, R. S. and de Moor, O.: *Algebra of programming*, Prentice-Hall, Inc. (1997).
6) Chin, W., Takano, A. and Hu, Z.: Parallelization via Context Preservation, *International Conference on Computer Languages*, pp. 153–162 (1998).
7) Cole, M.: *Algorithmic skeletons : A structured approach to the management of parallel computation*, Research Monographs in Parallel and Distributed Computing, Pitman, London (1989).
8) Dhillon, I. S. and Modha, D. S.: A Data-Clustering Algorithm on Distributed Memory Multiprocessors, *Large-Scale Parallel Data Mining, Lecture Notes in Artificial Intelligence*, pp. 245–260 (2000).
9) GETA: Generic Engine for Transposable Association.

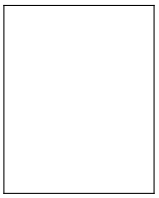
Available at http://geta.ex.nii.jp/.
10) Grama, A., Gupta, A., Karypis, G. and Kumar, V.: *Introduction to Parallel Computing*, Addison-Wesley, second edition (2003).
11) Han, E.-H., Karypis, G. and Kumar, V.: Scalable parallel data mining for association rules, *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, ACM Press, pp. 277–288 (1997).
12) Hartigan, J. A.: *Clustering Algorithms*, John Wiley & Sons, Inc. (1975).
13) Jeuring, J.: *Theories for Algorithm Calculation*, PhD Thesis, Utrecht University (1993). Parts of the thesis appeared in the Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics.
14) Li, X., Jin, R. and Agrawal, G.: A Compilation Framework for Distributed Memory Parallelization of Data Mining Algorithms, *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IEEE Computer Society, p. 7.1 (2003).
15) Miller, R.: *Two Approaches to Architecture-Independent Parallel Computation*, PhD Thesis, Computing Laboratory, Oxford University (1994).
16) O'caml: Objective Caml.
Available at http://www.ocaml.org/.
17) Pettorossi, A. and Proietti, M.: Rules and Strategies for Transforming Functional and Logic Programs, *ACM Computing Surveys*, Vol. 28, No. 2, pp. 360–414 (1996).
18) Salton, G. and Buckley, C.: Term-Weighting Approaches in Automatic Text Retrieval., *Inf. Process. Manage.*, Vol. 24, No. 5, pp. 513–523 (1988).
19) Takano, A., Niwa, Y., Nishioka, S., Iwayama, M., Hisamitsu, T., Imaichi, O. and Sakurai, H.: Associative information access using DualNavi, *Kyoto International Conference on Digital Libraries 2000*, pp. 285–298 (2000).
20) Wise, D. S.: Representing Matrices as Quadtrees for Parallel Processors, *Information Processing Letters*, Vol. 20, No. 4, pp. 195–199 (1984).
21) Xu, D. N., Khoo, S.-C. and Hu, Z.: PType System: A Featherweight Parallelizability Detector., *Proceedings of APLAS*, pp. 197–212 (2004).
22) 江本健斗, 胡振江, 筧一彦, 武市正人: 二次元配列上の構成的並列スケルトンの実現, 日本ソフトウェア科学会第 21 回大会 (2004).
23) 松田一孝, 西岡真吾, 胡振江, 武市正人: 階層的分割による並列連想計算, 日本ソフトウェア科学会第 21 回大会 (2004).

(平成 16 年 12 月 20 日受付)
(平成 ? 年 ? 月 ? 日採録)



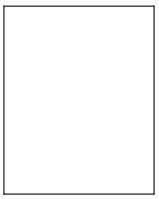
松田 一孝

1982 年生．2004 年東京大学工学部計数工学科卒業．同年同大学大学院情報理工学系研究科入学．アルゴリズムの導出，並列分散計算に興味を持つ．



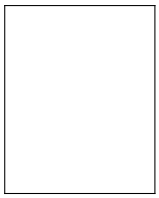
筧 一彦（正会員）

1997 年早稲田大学理工学部情報学科卒業．1999 年同大学大学院理工学研究科情報科学専攻修士課程修了，2002 年同博士課程修了．博士（情報科学）．1999 年から 2002 年まで日本学術振興会特別研究員．2002 年より東京大学大学院情報理工学系研究科助手．関数型言語やプログラム変換，アルゴリズムの導出に興味を持つ．日本ソフトウェア科学会，ACM 各会員．



胡 振江（正会員）

1966 年生．1988 年中国上海交通大学計算機科学系を卒業．1996 年東京大学大学院工学系研究科情報工学専攻博士課程修了．同年日本学術振興会特別研究員を経て，1997 年東京大学大学院工学系研究科情報工学専攻助手，同年 10 月同専攻講師, 2000 年同専攻助教授．2001 年より東京大学大学院情報理工学系研究科助教授，同年 12 月より科学技術振興事業団さきがけ 21 研究者を兼任．博士（工学）．関数プログラミング，プログラム変換，アルゴリズムの導出，並列プログラミングなどに興味を持つ．日本ソフトウエア科学会，ACM 各会員



武市　正人（正会員）

1948 年生．1972 年東京大学工学部助手，講師，電気通信大学講師，助教授，東京大学工学部助教授，を経て 1993 年東京大学大学院工学系研究科教授（情報処理工学講座），2001 年より同大学大学院情報理工学系研究科教授，現在に至る．工学博士．プログラミング言語，関数プログラミング，言語処理システムの研究・教育に従事．日本ソフトウェア科学会，日本応用数理学会，ACM 各会員．