情報処理学会論文誌　プログラミング　Vol. 4　No. 4　1–11 (Sep. 2011)

正規表現マッチングの並列化とそのHadoopでの評価

松崎公紀[†1]　江本健斗[†2]　劉雨[†3]

正規表現は広く用いられており，文章が正規表現にマッチするかどうかの問合せ（クエリ）を効率的に実行することは重要である．これまで，正規表現マッチングを高速に行う逐次的な手法について多くの研究がある．正規表現マッチングを並列に行う方法についても研究があるが，その多くは，複数の文章に対するクエリの並列実行や，複数のクエリの並列実行というような自明な並列実行について扱うものである．一方で，巨大な1つの文章に対して1つのクエリを行う場合には，正規表現マッチングそのものを並列化する必要が発生する．本稿では，正規表現マッチングを並列化する手法について議論を行う．また，本稿で提案する正規表現の並列マッチングの計算効率を評価するため，Hadoopを用いて実験を行いその結果を報告する．Hadoopは，大規模分散データに対して効率的に処理を行うことができるMapReduceフレームワークのオープンソース実装である．

Parallelization of Regular Expression Matching and Its Evaluation on Hadoop

Kiminori Matsuzaki,[†1] Kento Emoto[†2] and Yu Liu[†3]

Regular expressions are now widely used and efficient implementation of regular expression matching is an important issue for efficient manipulation of data. There have been many studies for efficient implementation of regular expression matching. There have also been studies on parallel implementations of regular expression matching, but these implementations exploit parallelism only in executing a single query on multiple documents or in executing multiple queries on a single document. In this paper, we discuss a technique to parallelize a regular expression query itself. In other words, with this technique we can execute a regular expression query on a single document in parallel. We evaluate efficiency of regular expression matchings parallelized by the proposed method on the Hadoop; the Hadoop is an open-source implementation of the MapReduce framework that enables efficient processing of large-scale data.

1. はじめに

正規表現は，grepプログラムをはじめ，テキスト処理においてとても広く用いられている．そのため，正規表現が文書にマッチするかどうかを判定する処理を高速に行うことは重要である．正規表現のマッチングを効率的に行う方法については古くから研究がなされ，非決定性有限オートマトンをもとにバックトラックを行う方法，バックトラックを行わない方法，決定性有限オートマトンを使う方法の3つがよく用いられる．また，正規表現のマッチングを行うライブラリや処理系の実装も多く存在する[6),26)]．

非常に大規模なデータに対して効率的に処理を行うにあたって，並列化は重要である．たとえばgrepプログラムについて考えると，通常のテキストデータに対しては，行単位の処理が独立していることを利用した自明な並列化が有効である．実際，GoogleのMapReduce[7)]やそのオープンソース実装のHadoop[23),24)]では，そのような並列プログラムがサンプルとして用いられており，非常に大規模なデータに対して効率的にかつ容易に並列処理を行うことが可能である．また，1つのデータに対して複数のクエリを実行する場合にも，自明な並列化を行うことができる．

一方，1つのクエリを分割できないデータに対して行う場合には，既存のライブラリや処理系の方法では逐次的に行うしかない．たとえば，XMLデータのような木構造を扱う場合，簡単なクエリであれば正規表現を用いてシリアライズされたデータ上の処理として記述することができる．しかし，そのようなクエリの場合には，たとえデータを分散して持っていたとしても，その部分データ対して独立に処理を行うことは単純にはできない．すなわち，クエリを適切に並列化する必要がある．

計算パターンを利用した並列プログラミング（スケルトン並列プログラミング[18),21)]）の分野において，そのようなクエリの並列化についての研究があり，オートマトンの有限性に着目して並列に計算することができることは知られていた[22),27)]．しかし，逐次による正規表現マッチングにおける非決定性有限オートマトンや決定性有限オートマトンを利用した方法と，それらの並列計算の方法との関連や，並列化にかかるオーバヘッドなどについては著

†1 高知工科大学情報学群
School of Information, Kochi University of Technology
†2 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
†3 総合研究大学院大学情報学専攻
Department of Informatics, The Graduate University for Advanced Studies

者の知る限り十分に議論されていなかった．

本研究の貢献は大きく次の 2 点である．

- 並列計算パターンの 1 つであるリスト準同型とその並列化のための理論に基づいて，正規表現マッチングを行う逐次プログラムの並列化について議論を行う．
- 並列化された正規表現マッチングアルゴリズムについて，大規模分散データ処理のための標準的なフレームワークである Hadoop 上で性能評価を行う．

本稿の構成は以下のとおりである．2 章では，本稿で用いる表記法について説明した後，リスト準同型の定義およびリスト準同型を用いて並列化を行うための条件について述べる．3 章では，MapReduce プログラミングモデルおよび Hadoop について説明し，リスト準同型が MapReduce 上で実現できることを示す．4 章では，正規表現マッチングがリスト準同型を用いてどのように並列に計算できるか議論する．5 章で，Hadoop 上での実験結果を報告する．6 章では関連研究について述べ，7 章で本稿をまとめる．

2. リスト準同型とそれを利用した並列化

2.1 表　記　法

本稿では，関数型言語 Haskell に似た表記を利用する．本稿での関数の定義は，関数適用が空白で表現され，また，関数がカリー化されていることを除けば，数学的な関数の定義と同様に読むことができる．

関数適用は空白によって表現し，引数には括弧をつけない．たとえば，$f\ a$ は $f(a)$ を表す．関数適用はカリー化され左結合である．すなわち，$f\ a\ b$ は $(f\ a)\ b$ である．関数適用は結合順位が最も高いとする．一般的な二項演算子として，$\oplus$，$\otimes$，$\ominus$ などを用いる．二項演算子はセクション化によって関数として扱い，$a \oplus b = (a\oplus)\ b = (\oplus b)\ a = (\oplus)\ a\ b$ である．演算子 $\circ$ は関数合成を表し，$(f \circ g)\ x = f\ (g\ x)$ である．また，演算子 $\bullet$ は逆向きの関数合成を表し，$(f \bullet g)\ x = g\ (f\ x)$ である．恒等関数を id とする．

固定個の値をまとめたものを組と呼ぶ．組の要素は異なる型を持ってもよい．組はたとえば $(1, 0.1)$ のように表記する．同じ型 D の要素 k 個からなる組に対しては，その型を D^k とも書く．組の第 1 要素を返す関数を fst，組の第 2 要素を返す関数を snd とする．演算子 $\triangle$ は複数の関数をまとめ，組を返す新しい関数を作る．たとえば，$(f \triangle g)\ x = (f\ x, g\ x)$ である．

同じ型の要素からなる有限の列をリストと呼ぶ．リストはその要素を [と] で囲んで表現し，空リストは [] で表現する．演算子 $+\!\!+$ は 2 つのリストを連接するもので，たとえば $[1,5,2] +\!\!+ [3,4] = [1,5,2,3,4]$ である．型が α である要素から構成されるリストの型を $[\alpha]$ と表現する．

データの重複を許す集合をマルチセットと呼び，{ と } で囲んで表す．集合から重複を除いて足し合わせる処理を $\cup$ にて表す．

2.2 foldl 関数とリスト準同型

リスト上の関数の形として，*foldl* 関数とリスト準同型の 2 つを定義する．

定義 1（foldl 関数） リスト上の *foldl* 関数とは，二項演算子 $\ominus$ と値 e を受け取り，

$$\begin{aligned} foldl\ (\ominus)\ e\ [\,] &= e \\ foldl\ (\ominus)\ e\ (x +\!\!+ [a]) &= (foldl\ (\ominus)\ e\ x) \ominus a \end{aligned}$$

という計算を行うものである．□

foldl 関数において，その演算子 $\ominus$ は結合的である必要はない．

定義 2（リスト準同型） 関数 f と結合的な二項演算子 $\oplus$ を用いて，関数 h が

$$\begin{aligned} h\ [a] &= f\ a \\ h\ (x +\!\!+ y) &= h\ x \oplus h\ y \end{aligned}$$

と定義できるとき，関数 h はリスト準同型である．また，このとき，関数 h を $h = hom\ f\ (\oplus)$ と表す．□

これまで，高レベル並列プログラミングの分野においてリスト準同型に関する研究が多数行われてきた[5),8),9)]．ある関数 h がリスト準同型であるならば，関数 h は分割統治法によって並列に計算することができる．すなわち，ある長いリストに対する計算は，より短いリストに対する計算を並列に行った後で，それらの結果をまとめればよい．関数 f および演算子 $\oplus$ の計算時間がそれぞれ定数時間であるとき，長さ n のリストに対して p 個のプロセッサを用いて $O(n/p + \log p)$ の時間で $hom\ f\ (\oplus)$ を計算することができることが知られている[21)]．

2.3 リスト準同型を用いた並列化

foldl 関数で表された計算に対して，それがリスト準同型である，もしくはリスト準同型の関数を用いて計算できることを示すことができれば並列化できたことになる．実際，任意の *foldl* 関数は，次のようにリスト準同型を用いた計算として表現することができる．以下の変形において，関数合成が結合的であることを利用している．

$$
\begin{aligned}
&\mathit{foldl}\ (\ominus)\ e\ [a_1, a_2, \ldots, a_n] \\
&\quad = (\cdots((e \ominus a_1) \ominus a_2)\cdots) \ominus a_n \\
&\quad = (\ominus a_n)\ (\cdots((\ominus a_2)\ ((\ominus a_1)\ e))\cdots) \\
&\quad = ((\ominus a_n) \circ \cdots \circ (\ominus a_2) \circ (\ominus a_1))\ e \\
&\quad = ((\ominus a_1) \bullet (\ominus a_2) \bullet \cdots \bullet (\ominus a_n))\ e \\
&\quad = \mathit{apply}\ (\mathit{hom}\ (\lambda\, a.(\ominus a))\ (\bullet)\ [a_1, a_2, \ldots, a_n])\ e \\
&\qquad \textbf{where}\ \mathit{apply}\ f\ e = f\ e
\end{aligned}
$$

しかし，関数を用いればリスト準同型として表せることは，並列化された計算が効率的であることを単純には意味しない．たとえば，関数合成を関数クロージャとして実現した場合，$\bullet$ の計算は定数時間でできるものの，*apply* の計算においてリストの長さに比例する時間がかかってしまう．効率的に計算できることをいうためには，上記の *apply*，$\lambda a.(\ominus a)$，$\bullet$ に相当する部分を，計算時間を抑えることができる何らかの値を用いた計算で置き換える必要がある．以下では，そのような置き換えが可能となるような計算の条件を 2 つ示す．

1 つ目は，演算子 $\ominus$ の左側の引数の定義域および値域が有限である場合である．この場合は，その定義域の要素すべてについて，それらがどのように遷移するするかをすべて計算すればよい．

補題 3（定義域の有限性[25]）　集合 $\mathcal{D}$ が l 個の有限要素からなるとし，$\mathcal{D} = \{d_1, \ldots, d_l\}$ とする．演算子 $\ominus$ が，適当な $\mathcal{A}$ について型 $\mathcal{D} \to \mathcal{A} \to \mathcal{D}$ を持ち，e を $\mathcal{D}$ の要素とするとき，$\mathit{foldl}\ (\ominus)\ e$ はリスト準同型を用いて並列化できる．

証明　次の等式

$$\mathit{foldl}\ (\ominus)\ e\ \mathit{xs} = \mathit{apply}\ (\mathit{hom}\ g\ (\oplus)\ \mathit{xs})\ e$$

を満たす関数 *apply*，g，$\oplus$ は次のとおり与えられる．

$$
\begin{aligned}
&\mathit{apply}\ (c_1, c_2, \ldots, c_l)\ e = \textbf{case}\ e\ \textbf{of}\ d_1 \to c_1; \cdots; d_l \to c_l \\
&g\ a = (d_1 \ominus a, \ldots, d_l \ominus a) \\
&(c_1, \ldots, c_l) \oplus (c'_1, \ldots, c'_l) = (c''_1, \ldots, c''_l) \\
&\quad \textbf{where} \\
&\qquad c''_i = \textbf{case}\ c_i\ \textbf{of}\ d_1 \to c'_1; \cdots; d_l \to c'_l
\end{aligned}
$$

このとき，関数 *apply* は $\mathrm{O}(1)$，関数 g と演算子 $\oplus$ は $\mathrm{O}(l)$ の時間でそれぞれ計算できる．□

補題 3 の証明では **case** を使ったプログラムを示したが，手続型プログラミングを使った実装では集合の要素を整数に対応づけることで，配列のインデックスアクセスとして実装することができる．

次に，関数が組の値を計算し，その計算において半環を構成する 2 つの演算子が使用されている場合を考える．

定義 4（半環の上で線形な関数）　代数 $(\mathcal{D}, \oplus, \otimes)$ を半環とし，k を定数とする．型 $\mathcal{D}^k \to \mathcal{D}$ を持つ関数 f が適当な定数 $a_1, \ldots, a_k$ を用いて

$$f\ (d_1, \ldots, d_k) = (d_1 \otimes a_1) \oplus \cdots \oplus (d_k \otimes a_k)$$

と定義されるとき，関数 f は半環 $(\mathcal{D}, \oplus, \otimes)$ の上で線形な関数であるという．□

組を計算する関数が半環の上で線形な関数の組によって記述されるとき，その関数は半環の 2 つの演算子を用いた行列積によって表現することができる．この性質を利用することで，*foldl* 関数をリスト準同型によって並列化することができる．

補題 5（半環上の組[10),25)]）　代数 $(\mathcal{D}, \oplus, \otimes)$ を半環とし，k を定数とする．適当な $\mathcal{A}$ について，型 $\mathcal{D}^k \to \mathcal{A} \to \mathcal{D}^k$ を持つ演算子 $\ominus$ を考える．$(\ominus a) = (f_1\ a \mathbin{\triangle} \cdots \mathbin{\triangle} f_k\ a)$ としたとき，すべての関数 $f_j\ a$（$j \in [1, \ldots, k]$）が半環 $(\mathcal{D}, \oplus, \otimes)$ の上で線形であるとする．また，e を $\mathcal{D}^k$ 型の値とする．このとき，$\mathit{foldl}\ (\ominus)\ e$ はリスト準同型で並列化できる．

証明　具体的には，次のようにリスト準同型を用いて表される．

$$\mathit{foldl}\ (\ominus)\ e\ \mathit{xs} = (\mathit{hom}\ \mathit{makeMatrix}\ (\times)\ \mathit{xs}) \times e$$

ただし，$\times$ は半環 $(\mathcal{D}, \oplus, \otimes)$ の上で定義される行列積とし，$\mathit{makeMatrix}\ a$ は線形な関数 $f_j\ a$ の係数を並べた行列を作る関数とする．ここで，*makeMatrix* 関数は $\mathrm{O}(k^2)$ の時間で，リスト準同型で用いられる行列積は $\mathrm{O}(k^3)$ の時間で，最後の行列ベルトル積は $\mathrm{O}(k^2)$ の時間でそれぞれ計算できる．□

3. MapReduce プログラミングモデルと Hadoop

Google 社の MapReduce は，非常に大規模なデータに対して多数の計算機を用いて効率的に処理を行うためのフレームワークである[7)]．また，そのような大規模並列処理を容易に記述することができるプログラミングモデルも提供している．これまでに，いくつかの MapReduce フレームワークが実装されており，その 1 つである Hadoop は MapReduce フレームワークのオープンソース実装である．

MapReduce における処理の流れを，**図 1** に示す．入力データは適当なサイズに分割され，

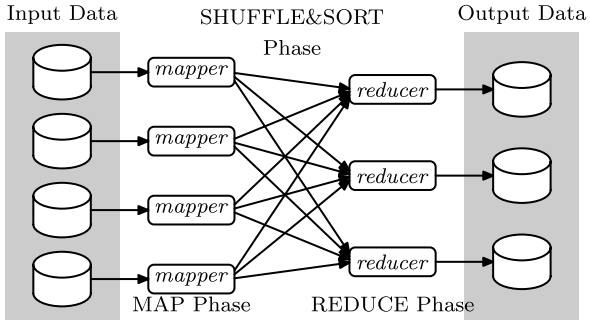

図 1　MapReduce における処理の流れ
Fig. 1　Three phases in MapReduce.

クラスタのノードに分散して置かれている．MapReduce の処理を通じて，データの各要素はキーと値の組として扱われる．MapReduce の処理は，大雑把にいうと，MAP [⋆1] フェーズ，SHUFFLE&SORT フェーズ，REDUCE フェーズの 3 つのフェーズから構成される．MAP フェーズでは，各キーと値の組に対して独立に処理が行われ，キーと値の組のリストが生成される．次に，SHUFFLE&SORT フェーズにおいて，同じキーを持つような値がまとめられる．最後に，REDUCE フェーズにおいて，同じキーを持つような値のリストに対して処理が行われ，各キーごとに結果が生成される．

より正確に議論を行うため，MapReduce における処理を定式化する．以下の定式化は，MapReduce の処理を Haskell によって定式化したもの[12] を参考にしている．MapReduce プログラミングにおいて，ユーザは 2 つの関数を定義する．

- 関数 *mapper*：この関数は，MAP フェーズにおいて用いられ，キーと値の組を 1 つ受け取り，キーと値の組のリスト（0 個でもよい）を生成する．一般に関数 *mapper* は，$mapper :: (k_1, v_1) \to [(k_2, v_2)]$ という型を持つ．
- 関数 *reducer*：この関数は，REDUCE フェーズにおいて用いられ，あるキーに対応する中間値のリストを受け取り，その値をまとめて結果を生成する．一般に関数 *reducer* は，$reducer :: (k_2, [v_2]) \to (k_3, v_3)$ という型を持つ．

⋆1 MapReduce の基本的な考え方は関数プログラミングに由来するといわれるが[7]，それゆえその処理の内容がよく誤解を招く．本稿では，関数プログラミングにおける関数と MapReduce における処理とを区別するため，MapReduce におけるフェーズには MAP と REDUCE を，そのフェーズにおいて使われるパラメータ関数には *mapper* と *reducer* を，関数プログラミングにおける高階関数には *map* と *reduce* をそれぞれ用いる．

MapReduce の処理は，次のプログラムによって与えられる．以下の定式化において，マルチセットを用いることによってデータが独立に並列処理されることを表現している．また，プログラムの 3 つの行は，それぞれ MapReduce の 3 つのフェーズに対応している．

$$
\begin{aligned}
&MapReduce :: ((k_1, v_1) \to [(k_2, v_2)]) \to ((k_2, [v_2]) \to (k_3, v_3)) \\
&\qquad\qquad \to \{(k_1, v_1)\} \to \{(k_3, v_3)\} \\
&MapReduce\ mapper\ reducer\ set\_kv \\
&\quad = \mathbf{let}\ set\_list\_kv = map_{\mathrm{S}}\ mapper\ set\_kv \\
&\qquad\quad set\_k\_list\_v = groupByKey\ set\_list\_kv \\
&\quad\ \mathbf{in}\ \ map_{\mathrm{S}}\ reducer\ set\_k\_list\_v
\end{aligned}
$$

上記のプログラムにおいて，関数 map_{S} はマルチセットの各要素に対して，引数として与えられる関数を適用するものである．関数 $groupByKey :: \{[(k, v)]\} \to \{(k, [v])\}$ は，キーと値の組のリストのマルチセットを入力にとり，同じキーを持つ値をまとめてリストとするものである．たとえば，*groupByKey* を $\{[(1, a), (2, z)], [(2, y), (1, b)]\}$ に適用した結果は，$\{(1, [a, b]), (2, [z, y])\}$ となる．ただし，値であるリストの要素間の順序は不定である．

以下，リスト準同型が MapReduce フレームワーク上で効率的に計算できることを示す．入力は，分割された部分データ（$[\alpha]$ 型）を値に，その部分が何番目か（*ID* 型）をキーに持つ組の集合とする．リスト準同型 $hom\ f\ (\oplus)$ の MapReduce 上での実装は次のプログラムで与えられる．以下のプログラムは，すでにある第 3 著者による定式化[13] を改良し，1 パスの MapReduce 計算により実現したものである．

$$
\begin{aligned}
&hom_{\mathrm{MR}} :: (\alpha \to \beta) \to (\beta \to \beta \to \beta) \to \{(ID, [\alpha])\} \to \beta \\
&hom_{\mathrm{MR}}\ f\ (\oplus) = getValue \circ MapReduce\ mapper\ reducer \\
&\quad \mathbf{where}\ mapper :: (ID, [\alpha]) \to [(ID, (ID, \beta))] \\
&\qquad\quad mapper\ (i, [as]) = [(0, (i, hom\ f\ (\oplus)\ as))] \\
&\qquad\quad reducer :: (ID, [(ID, \beta)]) \to (ID, \beta) \\
&\qquad\quad reducer\ (0, ibs) = \mathbf{let}\ bs = map\ snd\ (sortBy\ fst\ ibs) \\
&\qquad\qquad\qquad\quad \mathbf{in}\ (0, hom\ id\ (\oplus)\ bs) \\
&\qquad\quad getValue :: \{(ID, \beta)\} \to \beta \\
&\qquad\quad getValue\ \{(0, b)\} = b
\end{aligned}
$$

ただし，*map* はリストの各要素に対して引数の関数を適用したリストを返す関数である．まず，MAP フェーズでは，各部分データに対してリスト準同型を適用し，部分データの位置とリスト準同型の結果を組としたものを中間値として生成する．続く REDUCE フェーズでは，MAP フェーズで計算された部分結果のリストを受け取り，それを位置に基づいて並べ換え，リスト準同型の結合的な二項演算子でまとめる処理を行う．このプログラムの基本的な考え方は，ネストしたリストに対する次の有名な等式[2] に基づく．ここで，関数 *concat* をネストしたリストを平坦化する関数とする．

$$hom\ f\ (\oplus)\ (concat\ xss) = hom\ id\ (\oplus)\ (map\ (hom\ f\ (\oplus))\ xss)$$

2 章において，長さ n のリストに対して p 個のプロセッサを用いてリスト準同型が $O(n/p + \log p)$ で実現できることを述べた．上記の MapReduce を用いたリスト準同型の並列プログラムの場合，リストの長さを n，分割された部分データ数を m，プロセッサ数を p として，その計算量は $O(n/p + m \log m)$ となる．したがって，データの分割数は，負荷分散を行うことが可能な範囲で小さくなるようにすべきである．

4. 正規表現マッチングの並列化

正規表現のマッチングを行う方法として，非決定性オートマトン（以下，NFA）を用いる方法と決定性オートマトン（以下，DFA）を用いる方法がよく用いられる．逐次計算による正規表現マッチングにおいて，単純にマッチするかどうかを計算する場合には，多くの場合 DFA を用いる方が効率良く計算できる．しかし，NFA から DFA に変換する際に，最悪の場合 DFA の状態数が NFA の状態数の指数倍となることがあり，そのような場合には NFA を用いる方が効率良く計算できる．このように，NFA を用いる方法と DFA を用いる方法のそれぞれに長所短所がある．したがって，本章では，DFA を用いたマッチングおよび NFA を用いたマッチングについて，それぞれリスト準同型を用いた並列化を議論する．実際の場面でどちらが有効であると期待されるかについては，5 章の議論のところで著者の主張を述べる．

本章での並列化の議論の理解を助けるため，例として，正規表現 `(a|ac)*b(a|ac)*`を用いる．この正規表現のマッチングに対応する NFA の 1 つは，**図 2** に示されるものである．

4.1 DFA の並列化

正規表現のマッチングに対応する NFA から DFA を作成することで，単純なループによって正規表現のマッチングを行うことができる．DFA は，入力アルファベットの集合 Σ，状態の集合 Q_D，開始状態 $q_{0\mathrm{D}}$（$q_{0\mathrm{D}} \in Q_\mathrm{D}$），終了状態の集合 F_D（$F_\mathrm{D} \subset Q_\mathrm{D}$），および遷移関数 δ_D（$\delta_\mathrm{D} :: Q_\mathrm{D} \times \Sigma_\mathrm{D} \to Q_\mathrm{D}$）の 5 つ組 $(\Sigma, Q_\mathrm{D}, q_{0\mathrm{D}}, F_\mathrm{D}, \delta_\mathrm{D})$ で与えられる．例として用いる正規表現に対する DFA の場合，$\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$，$Q_\mathrm{D} = \{q_{0000}, q_{1000}, q_{1100}, q_{0010}, q_{0011}\}$，$q_{0\mathrm{D}} = q_{1000}$，$F_\mathrm{D} = \{q_{0010}, q_{0011}\}$ であり，δ_D は次の表の形で定義される．

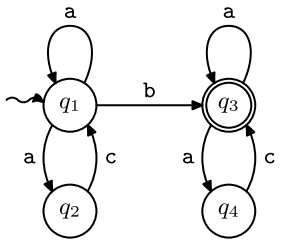

図 2　`(a|ac)*b(a|ac)*`に対応する NFA
Fig. 2　An NFA corresponding to `(a|ac)*b(a|ac)*`.

q	q_{0000}	q_{1000}	q_{1100}	q_{0010}	q_{0011}
$\delta_\mathrm{D}\ q\ \mathtt{a}$	q_{0000}	q_{1100}	q_{1100}	q_{0011}	q_{0011}
$\delta_\mathrm{D}\ q\ \mathtt{b}$	q_{0000}	q_{0010}	q_{0010}	q_{0000}	q_{0000}
$\delta_\mathrm{D}\ q\ \mathtt{c}$	q_{0000}	q_{0000}	q_{1000}	q_{0000}	q_{0010}

DFA $(\Sigma, Q_\mathrm{D}, q_{0\mathrm{D}}, F_\mathrm{D}, \delta_\mathrm{D})$ に対して，この DFA が入力文字列 *xs* を受理するかどうかを判定するプログラムは次のようになる．

$$\begin{aligned} &matchDFA\ q_{0\mathrm{D}}\ F_\mathrm{D}\ \delta_\mathrm{D}\ xs \\ &\quad = \mathbf{let}\ q = foldl\ (\ominus)\ q_{0\mathrm{D}}\ xs \\ &\qquad\qquad\qquad \mathbf{where}\ q \ominus x = \delta_\mathrm{D}\ q\ x \\ &\quad \mathbf{in}\ \ \mathbf{if}\ (q \in F_\mathrm{D})\ \mathbf{then}\ True\ \mathbf{else}\ False \end{aligned}$$

ここで，$\ominus$ の型は $\ominus :: Q_\mathrm{D} \to \Sigma \to Q_\mathrm{D}$ である．集合 Q_D の有限性と補題 3 より，上記のプログラムにおける *foldl* 関数はリスト準同型を用いて並列化できる．実際，補題 3 を上記の *foldl* 関数に適用すると次のプログラムを得る．ただし，$Q_\mathrm{D} = (q_1, q_2, \ldots, q_{|Q_\mathrm{D}|})$ とする．

$$
\begin{aligned}
&\mathit{matchDFA}\ q_{0\mathrm{D}}\ F_{\mathrm{D}}\ \delta_{\mathrm{D}}\ \mathit{xs}\\
&\quad = \mathbf{let}\ (p_1, p_2, \ldots, p_{|Q_{\mathrm{D}}|}) = \mathit{hom}\ g\ (\oplus)\ \mathit{xs}\\
&\qquad \mathbf{where}\ g\ x = (\delta_{\mathrm{D}}\ q_1\ x, \ldots, \delta_{\mathrm{D}}\ q_{|Q_{\mathrm{D}}|}\ x)\\
&\qquad\quad (p_1, \ldots, p_{|Q_{\mathrm{D}}|}) \oplus (p'_1, \ldots, p'_{|Q_{\mathrm{D}}|}) = (p''_1, \ldots, p''_{|Q_{\mathrm{D}}|})\\
&\qquad\qquad \mathbf{where}\ p''_i = \mathbf{case}\ p_i\ \mathbf{of}\ q_1 \to p'_1; \ldots; q_{|Q_{\mathrm{D}}|} \to p'_{|Q_{\mathrm{D}}|}\\
&\qquad q = \mathbf{case}\ q_{0\mathrm{D}}\ \mathbf{of}\ q_1 \to p_1; \ldots; q_{|Q_{\mathrm{D}}|} \to p_{|Q_{\mathrm{D}}|}\\
&\quad \mathbf{in}\ \ \mathbf{if}\ (q \in F_{\mathrm{D}})\ \mathbf{then}\ \mathit{True}\ \mathbf{else}\ \mathit{False}
\end{aligned}
$$

例の正規表現の場合，DFA のパラメータを上記のリスト準同型によるプログラムへ代入することによって，次のプログラムを得る．

$$
\begin{aligned}
&\mathit{matchDFAExample}\ \mathit{xs}\\
&\quad = \mathbf{let}\ (\_, p_{1000}, \_, \_, \_) = \mathit{hom}\ g\ (\oplus)\ \mathit{xs}\\
&\qquad \mathbf{where}\ g\ x = (\delta_{\mathrm{D}}\ q_{0000}\ x, \delta_{\mathrm{D}}\ q_{1000}\ x, \delta_{\mathrm{D}}\ q_{1100}\ x, \delta_{\mathrm{D}}\ q_{0010}\ x, \delta_{\mathrm{D}}\ q_{0011}\ x)\\
&\qquad\quad (p_{0000}, p_{1000}, p_{1100}, p_{0010}, p_{0011}) \oplus (p'_{0000}, p'_{1000}, p'_{1100}, p'_{0010}, p'_{0011})\\
&\qquad\qquad = (p''_{0000}, p''_{1000}, p''_{1100}, p''_{0010}, p''_{0011})\\
&\qquad\qquad \mathbf{where}\ p''_i = \mathbf{case}\ p_i\ \mathbf{of}\ q_{0000} \to p'_{0000}; q_{1000} \to p'_{1000};\\
&\qquad\qquad\qquad\qquad q_{1100} \to p'_{1100}; q_{0010} \to p'_{0010}; q_{0011} \to p'_{0011}\\
&\qquad \mathbf{in}\ \ \mathbf{if}\ (p_{1000} \in \{q_{0010}, q_{0011}\})\ \mathbf{then}\ \mathit{True}\ \mathbf{else}\ \mathit{False}
\end{aligned}
$$

DFA に基づく並列マッチングの計算コスト　補題 3 の証明の中での議論にあるとおり，補題 3 を適用して得られるリスト準同型において，結合的な演算子には定義域の大きさに比例する計算コストがかかる．ここで議論した DFA の並列化において，定義域の大きさは $|Q_{\mathrm{D}}|$ である．したがって，全体の並列プログラムの時間計算量は，$\mathrm{O}((n/p + \log p)|Q_{\mathrm{D}}|)$ となる．一方，逐次プログラムの時間計算量は $|Q_{\mathrm{D}}|$ によらず $\mathrm{O}(n)$ である．したがって DFA に基づく並列化では，リスト準同型において結合的な演算子を用いる必要性から，DFA の状態数に比例するオーバヘッドが存在することになる．

4.2　NFA の並列化

NFA を用いた正規表現のマッチングでは，バックトラック法を利用することが多い．しかし，到達可能な状態の集合を扱うことで，NFA をもとに 1 ループで計算することもできる．ここでは，その NFA に基づく 1 ループによるマッチングの並列化を議論する．

NFA は入力アルファベットの集合 Σ，状態の集合 Q_{N}，開始状態 $q_{0\mathrm{N}}$（$q_{0\mathrm{N}} \in Q_{\mathrm{N}}$），終了状態の集合 F_{N}（$F_{\mathrm{N}} \subset Q_{\mathrm{N}}$），および遷移関数 δ_{N}（$\delta_{\mathrm{N}} : Q_{\mathrm{N}} \times \Sigma \to \{Q_{\mathrm{N}}\}$）の 5 つ組 $(\Sigma, Q_{\mathrm{N}}, q_{0\mathrm{N}}, F_{\mathrm{N}}, \delta_{\mathrm{N}})$ で与えられる．例として用いる正規表現に対する NFA の場合，$\Sigma = \{a, b, c\}$，$Q_{\mathrm{N}} = \{q_1, q_2, q_3, q_4\}$，$q_{0\mathrm{N}} = q_1$，$F_{\mathrm{N}} = \{q_3\}$ であり，δ_{N} は次の表で定義される．

q	q_1	q_2	q_3	q_4
$\delta_{\mathrm{N}}\ q$ `a`	$\{q_1, q_2\}$	$\{\,\}$	$\{q_3, q_4\}$	$\{\,\}$
$\delta_{\mathrm{N}}\ q$ `b`	$\{q_3\}$	$\{\,\}$	$\{\,\}$	$\{\,\}$
$\delta_{\mathrm{N}}\ q$ `c`	$\{\,\}$	$\{q_1\}$	$\{\,\}$	$\{q_3\}$

NFA $(\Sigma, Q_{\mathrm{N}}, q_{0\mathrm{N}}, F_{\mathrm{N}}, \delta_{\mathrm{N}})$ に対して，この NFA が入力文字列 xs を受理するかどうかを判定するプログラムは次のようになる．ただし，以下において計算される値は重複を取り除いた集合である．

$$
\begin{aligned}
&\mathit{matchNFA}\ q_{0\mathrm{N}}\ \delta_{\mathrm{N}}\ F_{\mathrm{N}}\ \mathit{xs}\\
&\quad = \mathbf{let}\ \{p_1, p_2, \ldots, p_i\} = \mathit{foldl}\ (\ominus)\ \{q_{0\mathrm{N}}\}\ \mathit{xs}\\
&\qquad\quad \mathbf{where}\ \{p_1, p_2, \ldots, p_j\} \ominus x\\
&\qquad\qquad = \delta_{\mathrm{N}}\ p_1\ x \cup \delta_{\mathrm{N}}\ p_2\ x \cup \cdots \cup \delta_{\mathrm{N}}\ p_j\ x\\
&\quad \mathbf{in}\ \ \mathbf{if}\ (\{p_1, p_2, \ldots, p_i\} \cap F \neq \phi)\ \mathbf{then}\ \mathit{True}\ \mathbf{else}\ \mathit{False}
\end{aligned}
$$

ここで，計算される集合はその要素数がたかだか $|Q_{\mathrm{N}}|$ 個であることから，集合をビットベクタを用いて表現することができる．以下，$Q_{\mathrm{N}} = \{q_1, q_2, \ldots, q_{|Q_{\mathrm{N}}|}\}$ とする．

$$
\begin{aligned}
&\mathit{matchNFA}\ q_{0\mathrm{N}}\ \delta_{\mathrm{N}}\ F_{\mathrm{N}}\ \mathit{xs}\\
&\quad = \mathbf{let}\ q'_{0\mathrm{N}} = (p_1, p_2, \ldots, p_{|Q_{\mathrm{N}}|})\ \mathbf{where}\ p_i = \mathbf{if}\ (q_i = q_{0\mathrm{N}})\ \mathbf{then}\ 1\ \mathbf{else}\ 0\\
&\qquad (v_1, v_2, \ldots, v_{|Q_{\mathrm{N}}|}) = \mathit{foldl}\ (\ominus)\ q'_{0\mathrm{N}}\ \mathit{xs}\\
&\qquad \mathbf{where}\\
&\qquad (w_1, w_2, \ldots, w_{|Q_{\mathrm{N}}|}) \ominus x = (w'_1, w'_2, \ldots, w'_{|Q_{\mathrm{N}}|})\\
&\qquad\quad \mathbf{where}\ w'_i = ((q_i \in \delta_{\mathrm{N}}\ q_1\ x) \wedge w_1) \vee \cdots \vee ((q_{|Q_{\mathrm{N}}|} \in \delta_{\mathrm{N}}\ q_{|Q_{\mathrm{N}}|}\ x) \wedge w_{|Q_{\mathrm{N}}|})\\
&\quad \mathbf{in}\ (v_1 \wedge (q_1 \in F_{\mathrm{N}})) \vee \cdots \vee (v_{|Q_{\mathrm{N}}|} \wedge (q_{|Q_{\mathrm{N}}|} \in F_{\mathrm{N}}))
\end{aligned}
$$

ここで，$(\{1, 0\}, \vee, \wedge)$ は半環である．また，$\ominus$ の定義は半環 $(\{1, 0\}, \vee, \wedge)$ 上で線形な計算

$$((q_i \in \delta_{\mathrm{N}}\ q_1\ x) \wedge w_1) \vee \cdots \vee ((q_{|Q_{\mathrm{N}}|} \in \delta_{\mathrm{N}}\ q_{|Q_{\mathrm{N}}|}\ x) \wedge w_{|Q_{\mathrm{N}}|})$$

を並べたものとなっている．したがって，補題 5 を適用して NFA のマッチングをリスト準

同型を用いて並列化することができる．リスト準同型を用いたプログラムは次のようになる．ただし，*makeMatrix* x は上記の *foldl* 関数の演算子 $\ominus$ の計算における $(q_i \in \delta_{\mathrm{N}}\ q_j\ x)$ を行列として並べたものである．演算子 $\times$ は，半環 $(\{1,0\}, \vee, \wedge)$ の上で定義される行列乗算である．表現を簡潔にするため，行列の向きを補題 5 とは逆にしてある．

$$
\begin{aligned}
&\mathit{matchNFA}\ q_{0\mathrm{N}}\ \delta_{\mathrm{N}}\ F_{\mathrm{N}}\ \mathit{xs}\\
&\quad = \mathbf{let}\ q'_{0\mathrm{N}} = (p_1, p_2, \ldots, p_{|Q_{\mathrm{N}}|})\ \mathbf{where}\ p_i = \mathbf{if}\ (q_i = q_{0\mathrm{N}})\ \mathbf{then}\ 1\ \mathbf{else}\ 0\\
&\qquad\quad (v_1, v_2, \ldots, v_{|Q_{\mathrm{N}}|}) = q'_{0\mathrm{N}} \times (\mathit{hom}\ \mathit{makeMatrix}\ (\times)\ \mathit{xs})\\
&\qquad \mathbf{in}\ (v_1 \wedge (q_1 \in F_{\mathrm{N}})) \vee \cdots \vee (v_{|Q_{\mathrm{N}}|} \wedge (q_{|Q_{\mathrm{N}}|} \in F_{\mathrm{N}}))
\end{aligned}
$$

例の正規表現の場合，パラメータを代入することで，次のリスト準同型を用いた並列プログラムを得る．

$$
\begin{aligned}
&\mathit{matchNFAExample}\ \mathit{xs}\\
&\quad = \mathbf{let}\ (\_, \_, v_3, \_) = (1,0,0,0) \times (\mathit{hom}\ \mathit{makeMatrix}\ (\times)\ \mathit{xs})\\
&\qquad \mathbf{in}\ v_3
\end{aligned}
$$

ただし，*makeMatrix* は次のように定義される．

x	a	b	c
makeMatrix x	$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$

NFA に基づく並列マッチングの計算コスト　補題 5 の証明の中での議論にあるとおり，ベクタや行列の 1 辺のサイズが k であるとき，補題 5 を適用して得られるリスト準同型のプログラムの結合的な演算子には $\mathrm{O}(k^3)$ の計算コストがかかる．NFA の並列化において，$k = |Q_{\mathrm{N}}|$ であり，全体の並列プログラムの時間計算量は，$\mathrm{O}((n/p + \log p)|Q_{\mathrm{N}}|^3)$ となる．一方，逐次プログラムの計算は，各要素についてベクトル行列積を行うこととなるため，その時間計算量は $\mathrm{O}(n|Q_{\mathrm{N}}|^2)$ である．したがって NFA に基づく並列化では，リスト準同型において結合的な演算子を用いる必要性から，NFA の状態数に比例するオーバヘッドが存在することになる．

4.3 拡　　張

以上において，正規表現のマッチングが成功したかどうかを返す処理について議論した．正規表現を利用する場面によっては，文書中のどの部分にマッチしたのかが必要となることがある．以下で，そのようなクエリをどのように処理できるのかについて，簡単に述べる．

文書の先頭からマッチを行い，指定した正規表現へのマッチングが成功した場所を求めたいとする．そのような方法として 2 つの方法がある．1 つは，scan（prefix-sums とも呼ばれる）という計算パターン[3] を利用するもので，もう 1 つはマッチングの処理において成功・不成功だけでなく成功した場所を同時に返す関数に拡張することである．リスト準同型 *hom* f $(\oplus)$ が与えられたとき，そのリスト準同型をすべての接頭部分列に適用する処理は並列計算パターン scan を用いて効率的に並列に計算できる[3]．したがって，DFA もしくは NFA のいずれかをもとにリスト準同型を用いた並列プログラムを得ることができれば，すべての接頭部分列に正規表現がマッチするかどうかを並列に計算することができる．一方，マッチした場所を同時に返す方法でも計算することができる．この場合，マッチが成功する可能性のある中間結果に対して，場所をすべて覚えておく必要がある．そのような場所が多数ある場合には，その場所を覚える計算コストが大きくなってしまうので正規表現の性質によって使い分けが必要となってくる．

正規表現にマッチした部分列の先頭および末尾の場所を求めたい場合には，上記の方法で正規表現にマッチする末尾の場所を求め，その各場所から逆向きに正規表現をマッチすることで，マッチする部分の先頭を求めることができる．特に，正規表現にマッチする末尾の場所が多数ある場合には，その並列性を利用できるため，逆向きのマッチは逐次処理にて実行することもできる．

5. Hadoop を用いた実験

5.1 実験の設定と結果

MapReduce フレームワークのオープンソース実装である Hadoop を用いて，正規表現マッチングを行う並列プログラムの性能の評価実験を行った．

実験には，1，2，4，8，12，16，24，32 ノードからなるクラスタを用いた．クラスタの各ノードは 2 つの Xeon CPU（Prestonia コア 2.4–2.8 GHz，または，Nocona コア 2.8 GHz）と 2 GB のメモリと IDE のハードディスクとを持ち，ノードはギガビットイーサネットで接続されている．Hadoop の実装には，バージョン 0.21.0 のものを用いた．

まず，例として用いた正規表現 `(a|ac)*b(a|ac)*`の場合について，DFA に基づく並列プ

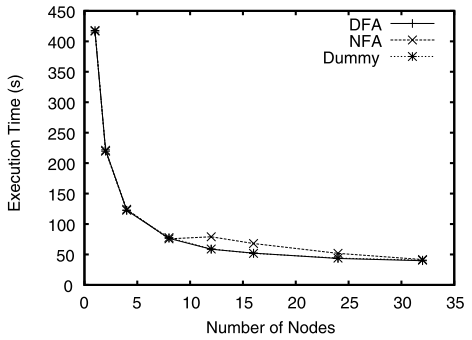

図 3　小さな正規表現のマッチングの実行時間
Fig. 3　Execution time for a small regular expression.

表 1　小さな正規表現のマッチングの実行時間（秒）
Table 1　Execution time for a small regular expression.

P	DFA	NFA	Dummy
1	415.3	416.8	417.8
2	221.8	221.1	219.6
4	122.6	124.2	122.6
8	76.5	75.7	77.8
12	58.7	79.1	58.9
16	52.2	68.0	51.9
24	43.5	51.9	44.1
32	39.7	41.7	40.7

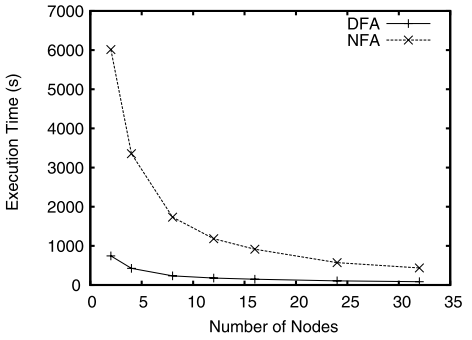

図 4　大きな正規表現のマッチングの実行時間
Fig. 4　Execution time for a large regular expression.

表 2　大きな正規表現のマッチングの実行時間（秒）
Table 2　Execution time for a large regular expression.

P	DFA	NFA
2	742.6	6,015.2
4	424.4	3,353.4
8	233.4	1,733.0
12	178.1	1,182.9
16	147.6	913.7
24	105.6	570.3
32	84.3	435.5

ログラムと NFA に基づく並列プログラムを作成し実行時間を計測した．また，データの読み込みにかかる時間について比較を行うため，データの長さをカウントするだけのプログラム（Dummy）を対照プログラムとして用いた．データサイズは 1 GB であり，Hadoop の HDFS により事前に 256 個の部分データに分割・分散して持っている．実行時間を**図 3** と**表 1** に示す．NFA に基づいて並列化を行ったものは行列積の計算コストの分だけ少し遅いが，ほとんどの時間が Hadoop によるデータの読み込みにかかるため，全体として見ると結合的な演算子を用いることのオーバヘッドはほとんどなく，効率的に計算できている．

次に，オートマトンの状態数が大きくなった際の実行時間について評価するため，状態数が 1,000 である DFA，および，状態数が 16 である NFA に対して，本稿の並列化を適用して得られるプログラムの実行時間を計測した．データについては上記と同じ設定としてある．実行時間を**図 4** と**表 2** に示す．前の実験の場合に比べると実行時間は大きくなっているものの，ノード数に対する台数効果としてほぼ線形の速度向上が得られている．

ここで，図 3 と図 4 の実行時間を見比べると，DFA に基づく並列プログラムと比べて NFA に基づく並列プログラムの方が実行時間がより大きく変化しているように見える．そこで，オートマトンの状態数を変化させたときの並列化したプログラムの実行時間を計測した．データサイズは同様に 1 GB を 256 個に分割したもので，32 ノードのクラスタで実行した．DFA に基づく並列プログラムの実行時間を**図 5** と**表 3** に，NFA に基づく並列プログラムの実行時間を**図 6** と**表 4** に示す．これらの図に近似曲線を引くことで，DFA に基づく並列プログラムの実行時間が状態数の 1 次式で近似されること，NFA に基づく並列プログラムの実行時間が状態数の 3 次式で近似されることがそれぞれ確認できる．

5.2　議　　論

Hadoop を用いた実験の結果をふまえ，DFA の並列化と NFA の並列化のどちらが有効であると期待されるかについて，著者の主張を述べる．

正規表現が与えられたときに，正規表現に対応する NFA をまず求め，さらに NFA を DFA に変換するという手法が一般的である．ここで，最悪の場合，変換して得られた DFA の状態数が NFA の状態数の指数倍となる．しかし，実用的には，そのような DFA の状態数が非常に増大することはほぼ起きないことがすでに調べられている[1),15)]．特に，文献 15) では，lex プログラムにおいて正規表現から NFA と DFA を求めた結果，DFA の状態数が NFA の状態数の 10 倍を超えるものは 2.5%未満であり，40 倍を超えるものは 0.5%未満であることが報告されている．

このことから，正規表現マッチングの並列化を行う際には，多くの場合には，DFA の並列化を行う方が有効であると考える．ただし，DFA の状態数が非常に増大する場合には NFA の並列化の方が有利となることもあり（今回の実験では，NFA の状態数 12 と DFA の状態数 2^{12} のときがほぼ釣り合う），DFA の状態数の爆発が予想される場合に NFA の並列化を利用するという実装上の選択肢として NFA の並列化にも意味があると考える．

また，正規表現マッチングの最適化との関連についても述べる．オートマトンを用いたマッチングの最適化法として，オートマトンの状態数を最小化する手法がある[1)]．本研究の並列化は，与えられたオートマトンを開始地点として並列化を行うため，状態数を最小化し

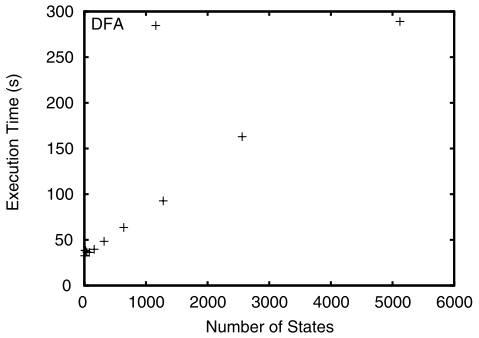

図 5　DFA に基づく並列プログラムの状態数に対する実行時間
Fig. 5　Execution time of the DFA-based parallel program with respect to the number of states.

表 3　DFA に基づく並列プログラムの状態数に対する実行時間（秒）
Table 3　Execution time of the DFA-based parallel program with respect to the number of states.

状態数	時間
5	34.2
10	40.1
20	39.8
40	38.2
80	37.9
160	41.3
320	49.9
640	65.1
1,280	94.3
2,560	164.5
5,120	290.7

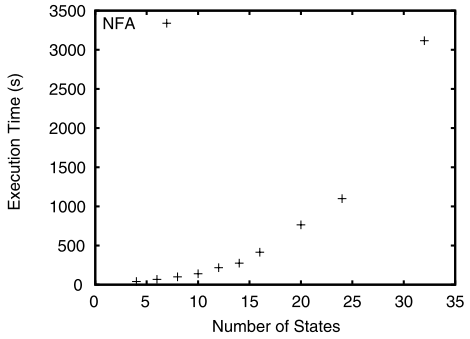

図 6　NFA に基づく並列プログラムの状態数に対する実行時間
Fig. 6　Execution time of the NFA-based parallel program with respect to the number of states.

表 4　NFA に基づく並列プログラムの状態数に対する実行時間（秒）
Table 4　Execution time of the NFA-based parallel program with respect to the number of states.

状態数	時間
4	40.3
6	67.4
8	100.5
10	139.6
12	216.8
14	275.0
16	414.3
20	763.9
24	1,099
32	3,116

た後に並列化すればより効率の良い並列プログラムを得ることができる．また，NFA におけるマッチングを高速化する手法として 32 個や 64 個のビット演算を同時に行う手法もあるが，この手法は NFA をもとに並列化したプログラムにおいても適用可能である．

一方，本稿で示した NFA に基づく並列プログラムに対しては，さらに「行列の要素がつねに 0 となる，もしくは，1 となる場合にその計算を省く」という最適化[14),17),20)] を行うことができる．ただし，この最適化を行うことで，どのくらい高速化が可能かを評価することは今後の課題である．

6. 関 連 研 究

6.1　クエリの並列化

クエリが複数個，もしくは，クエリを適用するデータが複数個ある場合には自明に並列化することができる．実際，Google の MapReduce の論文[7)] や Hadoop[23)] のサンプルでは，この自明な並列性をもとに並列 grep を実現している．また，XML 文書のような木構造の場合には，子の間の独立性を利用することで，独立な複数の部分データを作り，クエリ処理を行うこともできる[4),19)]．

一方，1 つのデータに対して 1 つのクエリを行う場合に，そのクエリ処理が並列にできることを示した文献もいくつか存在する．並列計算の理論的研究において，有限状態オートマトンが並列に計算できることは古くから研究されていた[11)]．しかし，実際の並列計算機環境において有効であるかどうかという観点での研究は少ない．Skillicorn[22)] および野村ら[27)] は，木データに対するクエリに対して，それらが木上の準同型およびそれを拡張した計算パターンによって表現でき，並列に計算できることを示している．そこでは，オートマトンの状態遷移の集合を組み合わせる計算が結合的であることを利用しており，本稿において DFA を用いて並列化する計算に対応するものである．一方，文献 25) では，再帰関数によって記述された性質を満たす部分データのうち値の合計が最大となるものを取得するクエリ（最大マーク付け問題）が並列に処理できることが示されている．そこでは，$+$ と max が半環をなすことに着目し，補題 5 を利用して並列化を行っている．

6.2　正規表現マッチングの実装

本稿では，正規表現のマッチングが並列に計算できることを示したが，それらをより高機能にかつより効率的に実装することは今後の課題である．これまで逐次処理の範囲で高速に正規表現マッチングを行うライブラリや処理系が多く開発されてきており，実装に関する具体的な工夫についてはそれらが参考になる．

たとえば，新屋ら[26)] は，継続を基本とする言語 CbC を用いて，DFA による非常に高速な正規表現マッチャを実装した．また，複数のマッチングアルゴリズムを選択することで安定した性能を出す正規表現ライブラリとして近年開発されているものに RE2[6)] などがある．また，Naghmouchi ら[16)] はデータアクセスが高速なハードウェアである GPU を利用して

高速に正規表現マッチングが実装できることを示している．

7. まとめと今後の課題

本稿では，リスト準同型による並列化を利用して，正規表現マッチングを並列化する方法について議論し，正規表現の並列マッチングの性能を Hadoop 上の実験により評価した．具体的には，NFA に基づくマッチングアルゴリズムと DFA に基づくマッチングアルゴリズムそれぞれに対して，系統的に並列プログラムを与える手法を示した．本稿で提案したリスト準同型を利用した並列化手法は，導出した結合的な演算子によるオーバヘッドが避けられないものの，プロセッサ数に対してスケーラビリティがあるという利点がある．実際，Hadoop 上で行った評価実験では，ノード数に対してほぼ線形に性能が向上することが確かめられた．

本稿では正規表現がリスト準同型に基いて並列に計算できることを示したが，並列に正規表現マッチングを行うようなライブラリもしくは言語処理系を作るには解決しなければならない点がいくつかある．まず，より表現力のある正規表現のクラスに対しても効率的に並列処理が可能かどうかの議論が必要である．次に，逐次的に行う計算の部分のさらなる高速化も必要である．本稿での並列プログラムでは，そのパラメータ関数として Java による素朴な実装を行った逐次関数を用いたが，その部分の最適化を行い高速化することは全体の高速化につながる．これついては，既存のライブラリや処理系の実装を参考にすることができる．また，このような高速化した実装によるさらなる評価を行うことも今後の課題の 1 つである．

謝辞　本研究の一部は，科学技術振興機構（JST）の戦略的国際科学技術協力推進事業（研究交流型）「日本—フランス（ANR）研究交流」の補助を受けて行われた．

参 考 文 献

1) Becchi, M. and Crowley, P.: Efficient Regular Expression Evaluation: Theory to Practice, *ANCS '08: Proc. 4th ACM/IEEE Symposium on Architectures for Networking and Communication Systems*, pp.50–59 (2008).
2) Bird, R.S.: *An Introduction to the Theory of Lists*, *Logic of Programming and Calculi of Discrete Design*, Broy, M. (Ed.), NATO ASI Series F, Vol.36, pp.5–42, Springer (1987).
3) Blelloch, G.E.: Scans as Primitive Operations, *IEEE Trans. Comput.*, Vol.38, No.11, pp.1526–1538 (1989).
4) Bordawekar, R., Lim, L. and Shmueli, O.: Parallelization of XPath Queries Using Multi-core Processors: Challenges and Experiences, *EDBT '09: Proc. 12th International Conference on Extending Database Technology: Advances in Database Technology*, Kersten, M., Novikov, B., Teubner, J., Polutin, V. and Manegold, S. (Eds.), pp.180–191, ACM (2009).
5) Cole, M.: Parallel Programming with List Homomorphisms, *Parallel Processing Letters*, Vol.5, No.2, pp.191–203 (1995).
6) Cox, R.: Regular Expression Matching can be Simple and Fast (but is Slow in Java, Perl, PHP, Python, Ruby, ...) (2007), available from ⟨http://swtch.com/~rsc/regexp/regexp1.html⟩.
7) Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *6th Symposium on Operating System Design and Implementation (OSDI2004)*, pp.137–150 (2004).
8) Gorlatch, S.: Systematic Efficient Parallelization of Scan and Other List Homomorphisms, *Euro-Par '96 Parallel Processing, 2nd International Euro-Par Conference, Proceedings, Volume II*, Bougé, L., Fraigniaud, P., Mignotte, A. and Robert, Y. (Eds.), Lecture Notes in Computer Science, Vol.1124, pp.401–408, Springer (1996).
9) Hu, Z., Iwasaki, H. and Takeichi, M.: Formal Derivation of Efficient Parallel Programs by Construction of List Homomorphisms, *ACM Trans. Prog. Lang. Syst.*, Vol.19, No.3, pp.444–461 (1997).
10) Kogge, P.M. and Stone, H.S.: A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations, *IEEE Trans. Comput.*, Vol.22, No.8, pp.786–793 (1973).
11) Ladner, R.E. and Fischer, M.J.: Parallel Prefix Computation, *J. ACM*, Vol.27, No.4, pp.831–838 (1980).
12) Lämmel, R.: Google's MapReduce Programming Model — Revisited, *Science of Computer Programming*, Vol.70, No.1, pp.1–30 (2008).
13) Liu, Y. and Hu, Z.: A Homomorphism-based MapReduce Framework for Systematic Parallel Programming, 第 13 回プログラミングおよびプログラミング言語ワークショップ（PPL 2011）(2011).
14) Matsuzaki, K., Hu, Z. and Takeichi, M.: Towards Automatic Parallelization of Tree Reductions in Dynamic Programming, *SPAA 2006: Proc. 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, Gibbons, P.B. and Vishkin, U. (Eds.), pp.39–48, ACM Press (2006).
15) Moscola, J., Lockwood, J., Loui, R.P. and Pachos, M.: Implementation of a Content-Scanning Module for an Internate Firewall, *FCCM '03: Proc. 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp.31–38 (2003).

16) Naghmouchi, J., Scarpazza, D.P. and Berekovic, M.: Small-ruleset Regular Expression Matching on GPGPUs: Quantitative Performance Analysis and Optimization, *Proc. 24th International Conference on Supercomputing, 2010*, Tsukuba, Ibaraki, Japan, June 2-4, 2010, Boku, T., Nakashima, H. and Mendelson, A. (Eds.), pp.337–348, ACM (2010).
17) Nistor, A., Chin, W.-N., Tan, T.-S. and Tapus, N.: Optimizing the Parallel Computation of Linear Recurrences Using Compact Matrix Representations, *Journal of Parallel and Distributed Computing*, Vol.69, No.4, pp.371–381 (2009).
18) Rabhi, F.A. and Gorlatch, S. (Eds.): *Patterns and Skeletons for Parallel and Distributed Computing*, Springer (2002).
19) Sarje, A. and Aluru, S.: A MapReduce Style Framework for Computations on Trees, *ICPP '10: Proc. 2010 39th International Conference on Parallel Processing*, pp.343–352, IEEE Computer Society (2010).
20) Sato, S. and Iwasaki, H.: Automatic Parallelization via Matrix Multiplication, *ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation* (*PLDI 2011*), pp.470–479 (2011).
21) Skillicorn, D.B.: *Foundations of Parallel Programming*, Cambridge International Series on Parallel Computation, Vol.6, Cambridge University Press (1994).
22) Skillicorn, D.B.: Structured Parallel Computation in Structured Documents, *Journal of Univarsal Computer Science*, Vol.3, No.1, pp.42–68 (1997).
23) The Apache Software Foundation: Welcome to Apache Hadoop! (2011), available from ⟨http://hadoop.apache.org/⟩.
24) White, T.: *Hadoop: The Definitive Guide*, O'Reilly Media (2009).
25) 松崎公紀，胡　振江，武市正人：リスト上の最大マーク付け問題を解く並列プログラムの導出，情報処理学会論文誌：プログラミング，Vol.49, No.SIG 3 (PRO 36), pp.16–27 (2008).
26) 新屋良磨，河野真治：動的なコード生成を用いた正規表現マッチャの実装，第 52 回プログラミング・シンポジウム (2011).
27) 野村芳明，江本健斗，松崎公紀，胡　振江，武市正人：木スケルトンによる XPath クエリの並列化とその評価，コンピュータソフトウェア，Vol.24, No.3, pp.51–62 (2007).

(平成 23 年 2 月 15 日受付)
(平成 23 年 6 月 13 日採録)

松崎　公紀（正会員）
1979 年生．2001 年東京大学工学部計数工学科卒業．2003 年同大学大学院情報理工学系研究科修士課程修了．2005 年同研究科博士課程中退．同年より同研究科助手，2007 年より助教，2009 年より高知工科大学情報学群准教授となり現在に至る．博士（情報理工学）．並列プログラミング，アルゴリズム導出等に興味を持つ．日本ソフトウェア科学会，ACM，IEEE，各会員．

江本　健斗（正会員）
1980 年生．2004 年東京大学工学部計数工学科卒業．2006 年同大学大学院情報理工学系研究科修士課程修了．2007 年同研究科博士課程中退．同年より同研究科研究員，2009 年より同研究科助教となり現在に至る．博士（情報理工学）．並列プログラミング，アルゴリズム導出等に興味を持つ．日本ソフトウェア科学会会員．

劉　　雨
1980 年生．2004 年中国西安交通大学計算機科学技術学科卒業．同年よりソフトウェアエンジニアとして勤務．2007 年来日，2009 年総合研究大学院大学情報学専攻博士課程入学．現在に至る．並列プログラミング，アルゴリズム導出等に興味を持つ．