日本ソフトウェア科学会第 28 回大会 (2011 年度) 講演論文集

並列化と実行時コード生成を用いた正規表現マッチングの高速化

新屋 良磨　光成 滋生　佐々 政孝

正規表現によるパターンマッチングは広く用いられており, これまでマッチング高速化のための様々な手法が研究されてきた. 正規表現を DFA に変換してマッチングを行う手法もその一つである. 本研究では二つの高速化手法を提案する. 一つは DFA を拡張し, マッチング対象となる文字列を複数に分割して並列マッチングを行う同時初期状態有限オートマトン (*Simultaneous Start-state Finite Automata*, **SSFA**) を提案する. 実際に SSFA を実装し, マルチコアマシン上での並列マッチングと状態数の評価を行い, その有用性を確認した. もう一つは与えられた正規表現からそれに対応するネイティブコードを実行時に直接生成する手法である. 我々の既存研究では, 正規表現に対応する DFA から C ソースコードを生成し, それをコンパイルする二段階の手法を用いてきた. それに対してこの手法は, 既存のコンパイラよりもきめ細かい最適化を行うことで, より高速なマッチングが可能になった.

1　はじめに

正規表現はシンプルかつ高速なパターンマッチングのための記法として, GNU grep などテキスト処理ツールや Web 上での大規模な検索 [2], ネットワーク上でのパケットフィルタリング [8] [6] など幅広く用いられている. そのため, 正規表現マッチングの高速化は重要な課題であり古くから研究されてきた [9]. 本研究で提案する高速化のための手法は以下の 2 点である.

1. 効率の良い並列マッチングのための SSFA の提案と実装
2. 正規表現に対応する DFA をシミュレートする x64 ネイティブコードを動的に生成

本論文は本章を含めて 7 章から構成される. 第 2 章は本論文で使用する正規表現や集合演算の表記法, オートマトンと基本的なパターンマッチングのアルゴリズムを説明する. 第 3 章ではマッチングを並列処理可能にするためのモデル SSFA を提案する. 第 4 章にてコード生成における各種最適化や既存手法との違いについて述べる. 第 5 章ではコード生成及び並列化についてベンチマークによる検証結果を報告する. 第 6 章で関連研究について述べ, 第 7 章は本研究のまとめとする.

Parallelization and Dynamic Code Generation for High-speed Regular Expression Matching.

Ryoma Shinya, Masataka Sassa, 東京工業大学情報理工学研究科数理・計算科学専攻, Department of Mathematical and Computing Sciences, Graduate School of Information Science and Engineering, Tokyo Institute of Technology.

Shigeo Mitsunari, サイボウズ・ラボ株式会社, Cybozu Labs, Inc.

2　正規表現によるパターンマッチング

2.1　表記法

本論文では, 以下に定義された演算のみを正規表現の演算として使用する.

連接　二つの言語 L と M の連接 (LM) は, L に属する列を一つとり, そのあとに M に属する列を連接することによってできる列全体からなる集合.

集合和　二つの言語 L と M の集合和 ($L|M$) は, L または M(もしくはその両方) に属する列全体からなる集合.

閉包　言語 L の閉包 ($L*$) とは, L の中から有限個の列を重複を許して取り出し, それらを連接してできる列全体の集合.

以上三つの正規表現における基本演算に加え,

- 言語 L と空文字 (ϵ) の集合和 $L|\epsilon$ を $L?$ で表す.
- 言語 L の中から m 個以上, n 個以下の列を重複を許して取り出した言語の連接,$L_1 \cdots L_m L_{m+1}? \cdots L_n?$ を $L\{m,n\}$ で表す. $m = n$ の場合は単に $L\{m\}$ とする.
- 文字 α, β の集合和 $\alpha|\beta$ を $[\alpha\beta]$ と表す. 長さ 1 の文字列の任意個の集合和はこの記法で纏めて表現する. また, /[0123456789]/のように文字が連続している場合/[0-9]/という略記法を用いる.
- 全ての文字の集合和を「.」で表す. これはどのような文字にもマッチする.

をそれぞれ糖衣構文として用いる. さらに本論文では正規表現と単純な文字列を区別するために正規表現を「/」で, 文字列を「“」「”」で囲みそれぞれ/Regex/, “String” と表記する. パターンマッチングは完全マッチ, つまり入力文字列全体が正規表現にマッチすることを前提とする.

また本論文では集合論の一般的な記法 [14] に従い, オートマトンに関する諸定義を行う. 特に, 集合 A についてその元の個数を集合 A の大きさと呼び $|A|$, A の部分集合全体の集合を冪集合と呼び 2^A で表す. また A から B への写像全ての集合を $\mathrm{Map}(A, B)$ で表す.

2.2 有限オートマトン (Finite Automata, FA)

正規表現におけるパターンマッチングは, 正規表現から等価な有限オートマトンを構成することによって行うことができる [1] [12] [2].

有限オートマトンとは有限個の状態で構成され, 入力を 1 文字読み次の状態に遷移 (状態遷移) することを繰り返し, 文字列を読み終えた時点で受理状態であればその文字列を「受理」し, そうでない場合「非受理」とする言語の判定を行うモデルある. 有限オートマトンには非決定性/決定性の性質を持つ NFA/DFA がある. 非決定性は状態遷移について複数の遷移先を許すことを意味し, 逆に決定性は遷移先が唯一であることを意味する. 非決定性は決定性の一般化であるため, 全ての DFA は NFA でもある. NFA は

NFA $N = (Q_N, \Sigma, \delta_N, q_{N0}, F_N)$

Q_N 状態の有限集合

Σ 入力文字の有限集合

$\delta_N : Q_N \times \Sigma \to 2^{Q_N}$ 遷移関数

$q_{N0} \in Q_N$ 初期状態

$F_N \subseteq Q_N$ 受理状態集合

で定義される. 長さ r の正規表現から等価な NFA N を直接構成することができ, その時の N の状態数は $|Q_N| = O(r)$ となる [1]. 同じく DFA は,

DFA $D = (Q_D, \Sigma, \delta_D, q_{D0}, F_D)$

Q_D 状態の有限集合

Σ 入力文字の有限集合

$\delta_D : Q_D \times \Sigma \to Q_D$ 遷移関数

$q_{D0} \in Q_D$ 初期状態

$F_D \subseteq Q_D$ 受理状態集合

で定義される. 正規表現と等価な DFA を得るには, 等価な NFA から以下の構成法を用いる.

アルゴリズム 1 NFA からの DFA 構成法

$Q_D, Q_{tmp} \subseteq 2^{Q_N}$
$Q_D \leftarrow \emptyset$
$q_{D0} \leftarrow \{q_{N0}\}$
$Q_{tmp} \leftarrow \{q_{D0}\}$
while($Q_{tmp} \neq \emptyset$) {
  pick up q_d from Q_{tmp}
  for all $a \in \Sigma$ {
    $q_{dnext} \leftarrow \bigcup_{q_n \in q_d} \delta_N(q_n, a)$
    $\delta_D(q_d, a) := q_{dnext}$
    if($q_{dnext} \notin Q_D$) {
      $Q_{tmp} \leftarrow Q_{tmp} \cup \{q_{dnext}\}$
    }
  }
  $Q_D \leftarrow Q_D \cup \{q_d\}$
  $Q_{tmp} \leftarrow Q_{tmp} \setminus \{q_d\}$
}
$F_D \leftarrow \{q_d \in Q_D | q_d \cap F_N \neq \emptyset\}$

DFA の各状態 Q_D は NFA の状態集合の部分集合であり, この構成法を**部分集合構成法** *(Subset Construction, Powerset Construction)* [1] [12] と言う. NFA N から構成した DFA D は常に

- $Q_D \subseteq 2^{Q_N}$

- $\delta_D(R,\alpha) = \bigcup_{r\in R} \delta_N(r,\alpha)$ for $(R,\alpha) \in Q_D \times \Sigma$
- N が言語 L を受理する時, かつその時に限り D は言語 L を受理する.

を満たし, $|Q_D| \leq |2^{Q_N}| = 2^{|Q_N|}$ の状態数で構成できる事がわかる. 実際には多くの正規表現は $O(|Q_N|^3)$ の状態数で DFA を作れることが知られている [1].

2.3 パターンマッチングのアルゴリズム

正規表現によるパターンマッチングのアルゴリズムは, 大きく

- NFA ベースのバックトラックを用いる方法
- NFA ベースのバックトラックを用いない方法 [9] [2]
- DFA ベースの方法

があり, それぞれ長さ n の文字列に対して最悪計算量は $O(2^n), O(n|Q_N|), O(n)$ となる [2] [1]. 本研究ではマッチングの計算量が最も低い DFA ベースのマッチングを採用した.

3 並列実行によるマッチングの高速化

正規表現のマッチングアルゴリズムの研究は 1960 年代から盛んに行われている [2] [9] [11] が, マッチングの並列化について, オートマトンの拡張や実装/性能評価まで踏み込んだ研究は著者の知る限り十分に無い. 本研究では最終的に NFA/DFA から**同時初期状態有限オートマトン (SSFA)** を構成することで並列度 p, 入力長 n に対しそれぞれ $O(n/p+p|Q_N|^2), O(n/p+p)$ の並列マッチングを実装した.

3.1 マッチングの並列化

マッチングの並列化を考察するにあたり, まず通常の NFA によるマッチングを考える. NFA においては状態遷移は非決定的に行われるので, 遷移可能な状態の集合について遷移を更新すれば良い. NFA N, 文字列 $w \in \Sigma^*$ が与えられた場合, w の長さを $|w|$, i 番目の文字を w_i で表すとして

アルゴリズム 2 NFA によるマッチング

$T \subseteq Q_N$

$$\mathrm{Match}(N,w) = \begin{cases} Accept & (T_{|w|} \cap F_N \neq \emptyset) \\ Reject & (otherwise) \end{cases}$$

$T_0 = \{q_{N0}\}$

$T_i = \bigcup_{q\in T_{i-1}} \delta_N(q, w_i) \quad (i = 1, \cdots, |w|)$ と定義することができる.

次にこのアルゴリズムを並列化することを考えてみる. 文字列 w について p 個のプロセスで並列マッチングを行うには, それぞれのプロセスは p_i に対して w を p 分割した部分文字列 $w^i \in \Sigma^*, w = w^1w^2\cdots w^p$ について状態遷移を並列実行する必要がある. しかし, アルゴリズム 2 のような通常の状態遷移を並列化しようとしても, 各プロセス p_i は直前のプロセス p_{i-1} の遷移結果 q_{i-1} に依存してしまうため単純には並列化できない.

そこで, 「NFA の全状態について, それぞれを初期状態とした場合の状態遷移」を**同時初期状態遷移 (SST)** と呼ぶ. SST を各プロセスがそれぞれの部分文字列に対して並列に実行し, 最終的にそれぞれのプロセスの結果を集計する. すなわち SST によって得られた全ての状態における遷移結果から, 実際の初期状態に対応する結果を選択することで並列マッチングを行うことができる. SST は初期状態からその遷移状態への写像 $SST : Q_N \to 2^{Q_N}$ で表現することができる. NFA N, 並列度 p について文字列 $w = w^1w^2\cdots w^p, w^i \in \Sigma^*$ が与えられた時,

アルゴリズム 3 NFA による並列マッチング

$R \subseteq Q_N, T : Q_N \to 2^{Q_N}$

$$\mathrm{PMatch}(N,w,p) = \begin{cases} Accept & (R_p \cap F_N \neq \emptyset) \\ Reject & (otherwise) \end{cases}$$

$R_0 = \{q_{N0}\}$

$R_i = \bigcup_{q\in R_{i-1}} T^i_{|w^i|}(q) \quad (i = 1, \ldots, p)$

$T^i_0(q) := \{q\}$ for $q \in Q_N$

$T^i_j(q) := \bigcup_{q'\in T^i_{j-1}(q)} \delta_N(q', w^i_j)$ for $q \in Q_N$
$(j = 1, \ldots, |w^i|)$

と定義できる. $T^i_{|w^i|}$ はそれぞれ分割文字列に対して並列に計算可能で, SST を計算するために NFA の全ての状態に対して遷移状態を計算するので, 計算量は $O((n/p)|Q_N|^3)$ となる ($|w| = n$ で w を p 等分した

場合). R_i では T_i によって得られた部分文字列に対する SST(写像) を初期状態から適用していくことで, q_{N0} を初期状態とした文字列終端での遷移状態の集合が求まる. R_i において適用及び和集合を求める計算量は $O(|Q_N|^2)$ となり, 並列マッチング (PMatch) 全体の計算量は $O((n/p)|Q_N|^3 + p|Q_N|^2)$ となる.

3.2 同時初期状態有限オートマトン (Simultaneous Start-state Finite Automata, SSFA)

アルゴリズム 3 の並列マッチングは, 状態遷移に対応する T 内で NFA の全状態について SST をその都度更新しているため, 入力長と NFA の状態数の二乗の積に比例した計算量が必要だった. しかし, 本節で説明する同時初期状態有限オートマトン **SSFA** を構成することで部分文字列に対する T の計算量を $O(n/p)$ にすることができる.

SSFA は本研究で提案する NFA/DFA を並列実行するための有限オートマトンの拡張モデルであり, 定義は以下のようになる.

SSFA $S = (Q_S, \Sigma, \delta_S, q_{S0}, q_{R0}, F_R)$

Q_S 状態の有限集合

Σ 入力文字の有限集合

$\delta_S : Q_S \times \Sigma \to Q_S$ 遷移関数

$q_{S0} \in Q_S$ 初期状態

q_{R0} 基となる NFA/DFA の初期状態

F_R 基となる NFA/DFA の受理状態集合

SSFA は NFA/DFA どちらからも構成することができる. NFA N から SSFA を得るには以下の構成法を用いる.

アルゴリズム 4 NFA からの SSFA 構成法

$Q_S, Q_{tmp} \subseteq \mathrm{Map}(Q_N, 2^{Q_N})$
$Q_S \leftarrow \emptyset$
$q_{S0}(q) := \{q\}$ for $q_n \in Q_N$
$Q_{tmp} \leftarrow \{q_{S0}\}$
while($Q_{tmp} \neq \emptyset$) {
  pick up q_s from Q_{tmp}
  for all $a \in \Sigma$ {
    $q_{snext}(q_n) := \bigcup_{q'_n \in q_s(q_n)} \delta_N(q'_n, a)$ for $q_n \in Q_N$
    $\delta_S(q_s, a) := q_{snext}$
    if($q_{snext} \notin Q_S$) {
      $Q_{tmp} \leftarrow Q_{tmp} \cup \{q_{snext}\}$
    }
  }
  $Q_S \leftarrow Q_S \cup \{q_s\}$
  $Q_{tmp} \leftarrow Q_{tmp} \setminus \{q_s\}$
}
$q_{R0} \leftarrow q_{N0}$
$F_R \leftarrow F_N$

本研究ではこの構成法を**対応構成法 (Correspondence construction)**[†1]と呼ぶ. さらに, DFA D から SSFA を構成する場合は,

アルゴリズム 5 DFA からの SSFA 構成法

$Q_S, Q_{tmp} \subseteq \mathrm{Map}(Q_D, Q_D)$
$Q_S \leftarrow \emptyset$
$q_{S0}(q) := q$ for $q_d \in Q_D$
$Q_{tmp} \leftarrow \{q_{S0}\}$
while($Q_{tmp} \neq \emptyset$) {
  pick up q_s from Q_{tmp}
  for all $a \in \Sigma$ {
    $q_{snext}(q_d) := \delta_D(q_s(q_d), a)$ for $q_d \in Q_D$
    $\delta_S(q_s, a) := q_{snext}$
    if($q_{snext} \notin Q_S$) {
      $Q_{tmp} \leftarrow Q_{tmp} \cup \{q_{snext}\}$
    }
  }
  $Q_S \leftarrow Q_S \cup \{q_s\}$
  $Q_{tmp} \leftarrow Q_{tmp} \setminus \{q_s\}$
}
$q_{R0} \leftarrow q_{D0}$
$F_R \leftarrow F_D$

とすることで SSFA を得ることができる. 本研究ではこの構成法を**写像構成法 (Mapping construction)** と呼ぶ[†2]. これらの構成法は, アルゴリズム 1 で説明した NFA から DFA を構成する部分集合構成法の自

†1 ある集合から別な集合の冪集合への写像を対応 (Correspondence) と呼ぶ. 対応は写像の一般化である.

†2 もちろん, これらは部分集合構成法の命名規則を踏襲している.

然な拡張となっている. 部分集合構成法における DFA の状態 Q_D は NFA の状態の部分集合 2^{Q_N} と対応しており, 対応構成法/写像構成法ではそれぞれ SSFA の状態 Q_S は写像 $\mathrm{Map}(Q_N, 2^{Q_N}), \mathrm{Map}(Q_D, Q_D)$ に対応している.

SSFA における受理の判定は通常の FA とは異なり並列動作のためのモデルで, 受理判定には SSFA 外部での操作が必要となる. それは前節で記述した並列マッチングアルゴリズムそのもので, NFA を基に構成した SSFA S_n から

アルゴリズム 6 NFA から構成した SSFA による並列マッチング

$$R \subseteq Q_N, T : Q_N \to 2^{Q_N}$$
$$\mathrm{PMatch}(S_n, w, p) = \begin{cases} Accept & (R_p \cup F_R \neq \emptyset) \\ Reject & (otherwise) \end{cases}$$
$$R_0 = \{q_{R0}\}$$
$$R_i = \bigcup_{q \in R_{i-1}} T^i_{|w^i|}(q) \quad (i = 1, \ldots, p)$$
$$T^i_0 = q_{S0}$$
$$T^i_j = \delta_S(T^i_{j-1}, w^i_j) \quad (j = 1, \ldots, |w^i|)$$

と表現でき, その計算量は $O(n/p + p|Q_N|^2)$ となる. DFA を基に構成した SSFA S_d での受理判定は, R を 1 状態として扱えば良く

アルゴリズム 7 DFA から構成した SSFA による並列マッチング

$$R \in Q_D, T : Q_D \to Q_D$$
$$\mathrm{PMatch}(S_d, w, p) = \begin{cases} Accept & (R_p \in F_R) \\ Reject & (otherwise) \end{cases}$$
$$R_0 = q_{R0}$$
$$R_i = T^i_{|w^i|}(R_{i-1}) \quad (i = 1, \ldots, p)$$
$$T^i_0 = q_{S0}$$
$$T^i_j = \delta_S(T^i_{j-1}, w^i_j) \quad (j = 1, \ldots, |w^i|)$$

となり, その計算量は $O(n/p + p)$ となる.

NFA から対応構成法を用いることによって得られる SSFA は

1. $Q_S \subseteq \mathrm{Map}(Q_N, 2^{Q_N})$
2. $Q_S \times \Sigma$ の元 (ϕ, α) に対し, $\delta_S(\phi, \alpha) = \phi'$ は $\phi'(q) \mapsto \bigcup_{q' \in \phi(q)} \delta_N(q', \alpha)$ for $q \in Q_N$
3. N が言語 L 受理する時, かつその時に限り S は言語 L を受理する.

構成基	Q_S	$\lvert Q_S \rvert$	構成法
NFA	$Map(Q_N, 2^{Q_N})$	$2^{\lvert Q_N \rvert^2}$	対応構成法
DFA	$Map(Q_D, Q_D)$	$\lvert Q_D \rvert^{\lvert Q_D \rvert}$	写像構成法

表 1 SSFA の NFA/DFA からの各構成法

を常に満たし, DFA から写像構成法を用いることによって得られる SSFA は

1. $Q_S \subseteq \mathrm{Map}(Q_D, Q_D)$
2. $Q_S \times \Sigma$ の元 (ϕ, α) に対し, $\delta_S(\phi, \alpha) = \phi'$ は $\phi'(q) \mapsto \delta_D(\phi(q), \alpha)$ for $q \in Q_D$ となる.
3. D が言語 L を受理するとき, かつその時に限り S は言語 L を受理する.

を常に満たす. SSFA は基となる NFA/DFA の初期状態を固定せず, 全状態とその遷移結果の写像を状態として扱う. 故に, **同時初期状態オートマトン** (*Simultaneous Start-state FA*) と命名した. 写像構成法, 対応構成法から SSFA の状態は写像そのものであり, $|\mathrm{Map}(A, B)| = |B|^{|A|}$ [14] から $|Q_S| \leq 2^{|Q_N|^2}, |Q_S| \leq |Q_D|^{|Q_D|}$ の状態数で構成できることがわかる. 表 1 に各構成法についてまとめた.

3.3 SSFA の状態数に関する考察

SSFA は NFA/DFA の状態数の指数関数的な状態数で構成できることを示した. DFA の状態数は最大で $2^{|Q_N|}$ となるが, 一般的な正規表現において $O(|Q_N|^3)$ 程度の状態数で構築することが知られている [1]. それでは SSFA の状態数は, 一般的な正規表現においてどのような状態数を取り得るのだろうか? ここでは, いくつかの正規表現について NFA/DFA と SSFA の状態数について比較することで考察を行う.

/$(w_0 w_1 \cdots w_n)$*/ のような正規表現 (ただし w_i はそれぞれ全て異なる文字), は SSFA の状態数が対応する最小の DFA の状態数 (=最小の NFA の状態数) の二乗に比例する. /(abc)*/に対応する最小の DFA を図 1, 対応 SSFA を図 2 に示す. また, この時の SSFA の各状態と DFA の状態との対応を表 2 に示す.

DFA/SSFA の状態数は正規表現のパターンによって大きく異なり, 平均的な状態数の見積りは困難で

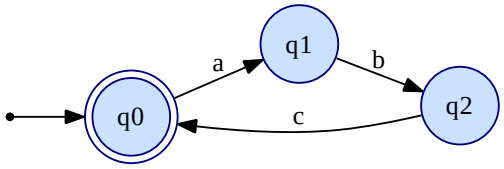

図 1　/(abc)*/を受理する最小 DFA

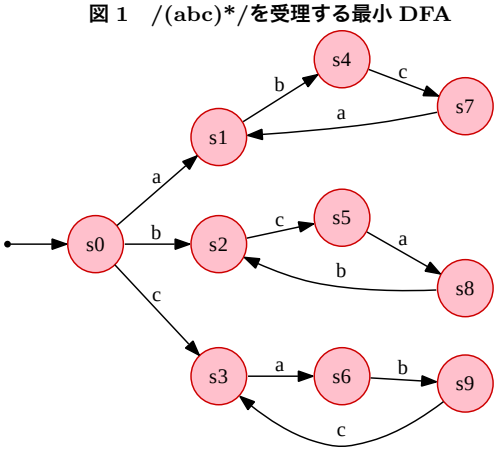

図 2　図 1 を並列動作させるための SSFA

Q_S	Mapping		
s_0	$q_0 \mapsto q_0$	$q_1 \mapsto q_1$	$q_2 \mapsto q_2$
s_1	$q_0 \mapsto q_1$	$q_1 \mapsto q_{dead}$	$q_2 \mapsto q_{dead}$
s_2	$q_0 \mapsto q_{dead}$	$q_1 \mapsto q_2$	$q_2 \mapsto q_{dead}$
s_3	$q_0 \mapsto q_{dead}$	$q_1 \mapsto q_{dead}$	$q_2 \mapsto q_3$
s_4	$q_0 \mapsto q_2$	$q_1 \mapsto q_{dead}$	$q_2 \mapsto q_{dead}$
s_5	$q_0 \mapsto q_{dead}$	$q_1 \mapsto q_0$	$q_2 \mapsto q_{dead}$
s_6	$q_0 \mapsto q_{dead}$	$q_1 \mapsto q_{dead}$	$q_2 \mapsto q_1$
s_7	$q_0 \mapsto q_0$	$q_1 \mapsto q_{dead}$	$q_2 \mapsto q_{dead}$
s_8	$q_0 \mapsto q_{dead}$	$q_1 \mapsto q_1$	$q_2 \mapsto q_{dead}$
s_9	$q_0 \mapsto q_{dead}$	$q_1 \mapsto q_{dead}$	$q_2 \mapsto q_2$

表 2　図 2 の SSFA と図 1 の DFA の同時初期状態遷移. *ここで, $a \mapsto b$ は $s_i(a) = b$ を表し,「a を初期状態とした場合, b に遷移する」を意味する.

ある. 本研究では典型的な正規表現のパターンから NFA/DFA/SSFA を構成し状態数を定量的に計測を行なったところ多くの正規表現に対して SSFA の状態数は DFA の状態数の二乗程度に収まる結果を得た. しかし, より定性的な平均状態数の評価や, 状態数が $|Q_S| = O(2^{|Q_N|^2}), |Q_S| = O(|Q_D|^{|Q_D|})$ となる正規表現については現段階で明らかでない. さらなる考察が必要と思われる.

4　並列化に関する補足

NFA から構成した SSFA S_n を用いたアルゴリズム 6 の並列マッチングにおいて, T は並列実行可能であるが R において R_i は R_{i-1} に依存している. そのため $p-1$ 回 R_i を順次計算する必要があり全体の計算量が $O((n/p+p)|Q_N|^3)$ となった. しかし, R における状態集合の更新を

$T : Q_N \to 2^{Q_N}$

$t1, t2 \in T$

$c \in (t2 \circ t1)(a) \Leftrightarrow \exists b[b \in t1(a) \wedge c \in t2(b)]$

で定義される写像の合成[†3]を用いることで

アルゴリズム 8 アルゴリズム 6 の写像合成版

$R, T : Q_N \to 2^{Q_N}$

$$\mathrm{PMatch}(S_n, w, p) = \begin{cases} Accept & (R(q_{R0}) \cup F_R \neq \emptyset) \\ Reject & (otherwise) \end{cases}$$

$R = T^p_{|w^p|} \circ T^{p-1}_{|w^{p-1}|} \circ \cdots \circ T^2_{|w^2|} \circ T^1_{|w^1|}$

$T^i_0 = q_{S0}$

$T^i_j = \delta_S(T^i_{j-1}, w^i_j) \quad (j = 1, \ldots, |w^i|)$

と並列マッチングを定義することができる. 写像の合成は結合的 [14] でそれぞれ並列計算可能である.

$T : Q_N \to 2^{Q_N}$

$t1, t2 \in T$

$$(t2 \circ t1)(a) := \bigcup_{b \in t1(a)} t2(b) \quad \text{for } a \in Q_N$$

で合成を計算することができ, その計算量は $O(|Q_N|^3)$ となる. さらに, DFA から構成した SSFA S_d を用いた並列マッチングでは

$T : Q_D \to Q_D$

$t1, t2 \in T$

$(t2 \circ t1)(a) := t2(t1(a))$

で定義される写像の合成を用いることで, 同様に

アルゴリズム 9 アルゴリズム 7 の写像合成版

$R, T : Q_D \to Q_D$

†3　厳密には「対応の合成」と言われる.

$$\mathrm{PMatch}(S_d, w, p) = \begin{cases} Accept & (R(q_{R0}) \in F_R) \\ Reject & (otherwise) \end{cases}$$
$$R = T^p_{|w^p|} \circ T^{p-1}_{|w^{p-1}|} \circ \cdots \circ T^2_{|w^2|} \circ T^1_{|w^1|}$$
$$T^i_0 = q_{S0}$$
$$T^i_j = \delta_S(T^i_{j-1}, w^i_j) \quad (j = 1, \ldots, |w^i|)$$

と並列マッチングを定義することができる. ここでの写像の合成は単に Q_D 全体についてそれぞれ二つの写像の像を計算すれば良く, 計算量は $O(|Q_D|)$ となる.

よって写像の合成を並列計算することで, SSFA による並列マッチング全体の計算量は NFA から構成した場合 $O(n/p + \log p \times |Q_N|^3)$, DFA から構成した場合 $O(n/p + \log p \times |Q_D|)$ とすることができる. なお本研究では並列度よりも NFA/DFA の状態数が大きい ($|Q_N| \geq |Q_D| > p$) マッチングを主に想定しているため状態数の次数が低いアルゴリズム 6,7 を実装に用いている.

5 コード生成を用いたマッチングの高速化

既存の DFA ベースの正規表現エンジンの実装では, 以下の C 言語風の擬似コードの様に DFA による状態遷移を繰り返し文と配列 (ルックアップテーブル) を用いて実装されることが多い.

```
bool FullMatchDFA(
  unsigned char *str,
  unsigned char *end) {
  int state = 0, next;
  while (str != end) {
    next = transition[state][*str++];
    if (next == DEAD_STATE) return false;
    state = next;
  }
  return IsAcceptState(state);
}
```

それぞれ使用されている変数は

str,end マッチング対象文字列の先頭/終端ポインタ

transition[ ] DFA の遷移関数 δ に相当する 2 次元配列

DEAD_STATE 死状態を表す定数

IsAcceptState() 受理状態の判定を行う関数

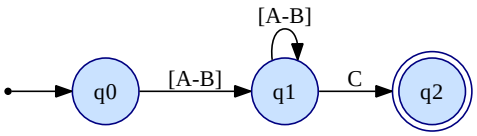

図 3 正規表現/[A-B]+C/に対応する DFA

を表す. 死状態とは, 遷移規則によって遷移が規定されていない時に使用される特別な状態で, その場合は以降どのような文字列を読み取っても遷移が受理状態にたどり着くことはないので途中でマッチングを終了することができる. 以上の実装を, 状態と遷移規則をデータ (変数と配列) で表現し状態遷移のシミュレートを行なってることから, 本論文では「**データ主導のマッチング**」と呼ぶことにする. データ主導の場合プログラム実行時に遷移規則を動的に構築することが容易で実装もしやすい. しかし, より高速な状態遷移や, 状態ごとに異なった命令を実行したい場合データ主導の実装では限界が出てくる.

これらの限界を克服するために, 本研究では実行時に DFA の状態遷移を機械語レベルで動的に生成する手法に着目した. 動的に生成されたコードは, 文字列への先頭/終端ポインタを受け取り遷移を行い終端に到達した時点で状態番号を返す. 状態番号による受理/非受理の判定は外部で行う. 例として正規表現/[A-B]+C/に対応する図 3 の DFA から生成されるコード C 言語レベルの記述は

```
q0: if (str == end) return 0; //状態番号
  if (*str++ - 'A' < 'B' - 'A' + 1) goto q1;
  else return -1;
q1: if (str == end) return 1;
  switch (*str++) {
    case 'A': case 'B': goto q1;
    case 'C': goto q2;
    default: return -1;
  }
q2: if (str == end) return 2;
  else return -1;
```

となる. 1 状態あたり 30byte 程度のネイティブコードが生成される.

正規表現から対応するプログラムを動的に生成する手法はこれまでにも提案されている [9] [16] が, 本研究ではより高速なコードを生成する手法として遷移規則の最適化と状態縮約による最適化を提案し, コード生成ライブラリである Xbyak を使用することでそれぞれの最適化をコンパイラを通さず直接機械語レベルで行った. 機械語を直接生成することが可能なので, コンパイラに依らず常に最適化されたコードを高速に生成することができる.

本章では, 説明をシンプルにするために生成されるコードは全て C 言語レベルの擬似コードで記述し, 最適化手法についてはアイディアを中心に説明する.

5.1 遷移規則の最適化

データ主導のマッチングで説明したように, 状態遷移規は入力文字種類数分 (8bit なら 256) の要素を持つ配列のルックアップで表現することができる.

しかしある状態において以下の条件を満たす時, 遷移規則を一つの条件分岐で表現することができる.

1. 遷移先が死状態を含めて二つ以下で, 一つの遷移先に対する遷移文字が 1 文字の場合.
2. 遷移先が死状態を含めて二つ以下で, 一つの遷移先に対する遷移文字が連続している場合.

条件 1 の場合は遷移文字を'c' とすると

```
if (*str++ == 'c') goto 遷移先 1;
else goto 遷移先 2;
```

条件 2 の場合は連続している遷移文字の中で最も大きい文字を upper, 最も小さい文字を lower とすると

```
if (*str++ - lower < upper - lower + 1)
  goto 遷移先 1;
else goto 遷移先 2;
```

と表現することができ, 例えば正規表現/[0-9]/に対応する遷移規則などで適用できる. このとき lower に '0', upper に '9' が入る. なお, 条件 2 の最適化において演算は符号なし整数として行う必要がある.

これらの最適化規則を**遷移規則の最適化**と呼び, 規則に当てはまらない遷移規則についてはデータ主導のマッチングと同じくルックアップテーブルを用いることで表現できる. しかしテーブルを参照する場合,

1. テーブルを参照するためにデータ領域へのアクセスが必要になる.
2. 必ずジャンプ命令の実行を伴う
3. 分岐先が複数あり予測が困難である.

と実行効率面で劣る場合がある. 特に 2 に関して, 最適化を行なった場合は分岐予測機構などハードウェア的支援を得やすい. さらに, 遷移規則最適化条件を満たした場合次節で説明する状態縮約による最適化を試みることができる.

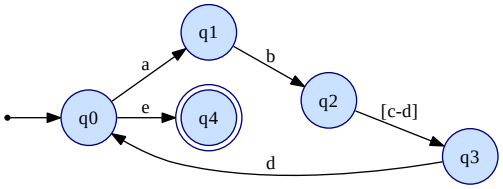

図 4 縮約最適化前

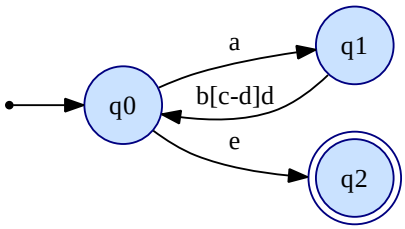

図 5 縮約最適化後

5.2 DFA 状態の縮約による最適化

特定の遷移規則によっては, 条件分岐によるジャンプ命令や文字列ポインタのインクリメント, 文字列の終端検査も最適化により消去することができる.

たとえば, 図 4 の DFA は図 5 のように, DFA の意味を変えることなく状態を纏めることができ, これを**状態の縮約**と呼ぶ. 矢印上の文字は遷移文字を表し, 前節で説明した遷移規則最適化に適合する遷移規則は 1 本の矢印に纏めることができる.

コード生成において, 各状態ごとに

- 文字ポインタの終端検査
- 文字ポインタのインクリメント
- テーブルルックアップ (switch) もしくは条件分岐
- ジャンプ命令の実行

それぞれの処理を行っていた. しかし, 縮約した状態ではこれらの命令を最適化によって纏めることができる. 図 4, 図 5 に対応するコードはそれぞれ

```
q0: /* 縮約前コード */
  if (str == end) return 0;
  switch (*str++) {
    case 'a': goto q1;
    case 'e': goto q4;
    default: return -1;
  }
q1:
  if (str == end) return 1;
  if (*str++ == 'b') goto q2;
  else return -1;
q2:
  if (str == end) return 2;
  if (*str++ - 'c' < 'd' - 'c' + 1) goto q3;
  else return -1;
q3:
  if (str == end) return 3;
  if (*str++ == 'd') goto q0;
  else return -1;
q4:
  if (str == end) return 4;
  else return -1;
```

```
q0: /* 縮約後コード */
  if (str == end) return 0;
  switch (*str++) {
    case 'a': goto q1;
    case 'e': goto q2;
    default: return -1;
  }
q1:
  if (str + 2 >= end) {
    if (str == end) return 1;
    else goto q1_; /* 注釈 */
  }
  if (str[0] != 'b') return -1;
  if (!(str[1]-'c'<'d'-'c'+ 1)) return -1;
  str += 3;
  if (str[-1] == 'd') goto q0;
  else return -1;
q2:
  if (str == end) return 4;
  else return -1;
```

のようになる.

最適化コードの場合, 文字列の終端検査と文字列ポインタのインクリメントが 1 命令に纏められている. また'b' や'c' での条件分岐において, 偽となる部分に次の文字比較コードを展開することで, ジャンプを伴わないハードウェア的に効率良く実行できる機械語を生成することができる. コメント「/* 注釈 */」でここに記述していない状態 q1_ にジャンプしているが, これは縮約によってまとめられた状態の先頭で文字列ポインタの終端をまとめて比較しており, 残りの文字列が縮約された状態の数よりも少ない場合は縮約された状態のいずれかで遷移が止まることとなる. 正しい状態番号を返すために, 縮約される前の状態へのコードへジャンプすることでこれを補っている. 注釈の部分は実装の都合が大きく, 本質的ではないためコードから省いた.

コード生成プログラムの実装しやすさや生成されるコードのサイズと実行効率を考慮し, 状態縮約最適化を行う条件を以下のように定義している.

1. 遷移規則の最適化が適用できる.
2. 死状態, 受理状態は縮約の対象とならない.
3. 遷移先が二つの場合, 一方が死状態である.
4. 遷移先は受理状態でなく, かつその状態の遷移元が唯一である.

条件 1 に関してはコードの展開に必要な条件であり, 状態縮約最適化の必須条件である. 前節で説明した遷移規則が適用される場合, テーブルジャンプから一つの条件分岐命令に置き換えることができ, その場合遷移先コードを展開することができる.

条件 2~4 は展開コードのサイズや実行性能に関する制約条件であり, 必須条件ではない. この条件については実装上の制約や都合が大きいので, 説明は省略する.

6 評価

6.1 評価方法

本章では 3 章で提案したコード生成最適化, 及び 4 章で提案したマッチングの並列化について, マッチング速度についてマルチコア環境でのベンチマークによる評価を行う. なお, それぞれのベンチマークでは正規表現に対して文字列全体がマッチするような文字列をメモリ上に同一プロセス内で生成しており, マッチングプログラムは文字列を全て読み込む. 実行時間は Intel の x86 系 CPU で使用可能な

rdtsc 命令によるクロック数を用い, 初期キャッシュミスやスケジューリングなどの外因を最小にするため同一プロセス内で初回実行時間を除く 10 回分の実行時間の最速値を採用している. また, スループットはマッチング時間と入力サイズのみから求めており コード生成時間は含んでいない. ベンチマークは全て SpeedStep/TurboBoost [7] を無効にした Intel Core i7-980X (3.33GHz, 6 物理コア,12 スレッド), 24GB DDR3-SDRAM (1333MHz) を搭載したマシン上で行い, 並列化には boost::thread を使用した (今回の実験環境 (Linux) では pthread の wrapper となる).

ベンチマークの対象となるプログラム名とエンジンの種類を以下に示す.

RE2 Goole RE2
O0 データ主導 (コード生成しない)
O1 コード生成
O2 コード生成+遷移規則最適化
O3 コード生成+遷移規則最適化+状態縮約最適化
Read 文字列を読むだけの指標プログラム.

6.2 コード生成を用いたマッチング

2 章で説明したデータ主導のマッチング実装と 3 章で説明したコード生成及び最適化を適用したマッチング実装, さらに既存の正規表現ライブラリから同じく DFA ベースのマッチングを行う Google RE2 [5] を対象に, それぞれコード生成 (**Codegen**)/マッチング (**Matching**) それぞれにかかったクロックサイクルでベンチマークをとり評価を行った. 結果を表 3, 表 4 に示す. ここで, コード生成時間は正規表現から DFA を構築する時間も含んだ時間とする.

2 章で述べたように DFA によるマッチングは入力文字列の長さにのみ依存するが, コード生成の最適化は正規表現に依存する. 以上の理由から, ここでは最適化が適用しやすい正規表現とそうでない正規表現の 2 パターンにおいて 1GB のテキストを対象にベンチマークを行なった.

表 3 から, データ主導のマッチング (O0) に対してコード生成版のマッチング (O1,O2,O3) は 3 倍程度高速に, さらに正規表現によっては最適化が遷移規則最適化と状態縮約最適化 (O3) によってデータ主導のマッチング (O0) に比べ 6 倍の速度が出ていることがわかり, 最適化の効果が高いことが見て取れる. 表 4 は最適化が効かない正規表現の実行例で, 連続しない遷移規則が繰り返し現れる正規表現を用いている. この場合は最適化の効果はなく (実際, 生成されるコードは同一) O1,O2,O3 それぞれ速度は変わらずデータ主導のマッチング (O0) に比べて 3 倍程度高速になっている. Google RE2 では on-the-fly な DFA の構築 [2] [4] や省メモリのため遷移テーブルを間接参照するなど工夫を施しており, 本実装によるシンプルなデータ主導のマッチング (O0) に比べて 2 倍ほど遅い結果

正規表現:/(0123456789)*/, 入力:1GB

Engine	Codegen	Matching	Throughput
RE2	54668	12630176840	0.263GB/sec
O0	190184	7414917052	**0.449GB**/sec
O1	360336	2188106432	1.521GB/sec
O2	398180	2669990652	1.247GB/sec
O3	423684	1225896300	**2.716GB**/sec
Read		551121603	6.042GB/sec

表 3 最適化が効く正規表現によるベンチマーク.
Codegen, Matching の単位はクロックサイクル

正規表現:/(([02468][13579]){5})*/ 入力:1GB

Engine	Codegen	Matching	Throughput
RE2	107256	12601966704	0.264GB/sec
O0	236960	7417497860	0.448GB/sec
O1	404688	2188165296	**1.521GB**/sec
O2	406732	2188207868	**1.521GB**/sec
O3	418368	2188205292	**1.521GB**/sec
Read		551121603	6.042GB/sec

表 4 最適化が効かない正規表現によるベンチマーク.
Codegen, Matching の単位はクロックサイクル

Num of States	O0 Codegen	O3 Codegen
16 (n = 3)	0.003sec	0.003sec
32 (n = 4)	0.008sec	0.008sec
64 (n = 5)	0.020sec	0.021sec
128 (n = 6)	0.050sec	0.051sec
256 (n = 7)	0.122sec	0.116sec
512 (n = 8)	0.289sec	0.290sec
1024 (n = 9)	0.676sec	0.683sec
2048 (n = 10)	1.568sec	1.583sec

表 5 /.*a.{n}/に対する状態数とコード生成時間

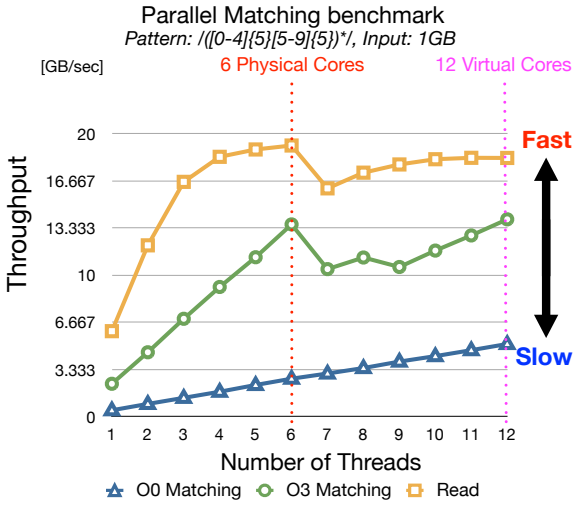

図 6　大きなテキストに対する並列マッチング．この時の DFA の状態数は 10 個, SSFA の状態数は 109 個.

正規表現: /(([0-4]{5}[5-9]{5}))*/, 入力: 1GB

thread	O0 Matching	O3 Matching	Read
1	0.449GB/sec	2.321GB/sec	6.041GB/sec
2	0.897GB/sec	4.550GB/sec	12.11GB/sec
3	1.327GB/sec	6.917GB/sec	16.60GB/sec
4	1.759GB/sec	9.167GB/sec	18.37GB/sec
5	2.236GB/sec	11.27GB/sec	18.89GB/sec
6	2.681GB/sec	**13.59GB**/sec	19.17GB/sec
7	3.047GB/sec	10.43GB/sec	16.14GB/sec
8	3.434GB/sec	11.25GB/sec	17.25GB/sec
9	3.891GB/sec	10.58GB/sec	17.83GB/sec
10	4.276GB/sec	11.74GB/sec	18.20GB/sec
11	4.698GB/sec	12.81GB/sec	18.30GB/sec
12	**5.134GB**/sec	**13.95GB**/sec	18.29GB/sec

表 6　図 6 に対応するスループット

となった.

次に, 本実装でのコード生成速度についての考察を行う. 正規表現/.*a.{n}/は等価な DFA の状態数が $O(2^n)$ で増加する性質を持つ. これを利用して, 状態数に対する O0,O3 でのコード生成時間を計測した結果を表 5 に示す. コード生成速度はそれぞれ 1500 状態/sec 程度であることが表 5 から読み取れる. また, O0/O3 両者の差が非常に小さいことからコード生成のオーバーヘッドが小さいことがわかる.

6.3　SSFA を用いた並列マッチング

6.3.1　大きなテキストに対するベンチマーク

4 章で説明した SSFA を構築し, 1GB のテキストに対してデータ主導マッチング (O0) 及びコード生成+遷移状態最適化+状態縮約最適化を適用したマッチング (O3) それぞれについて, 1~12 並列度 (スレッド) のベンチマーク結果を図 6 及び表 6 に示す. なお, 指標プログラムとして文字を読むだけのプログラム Read もテキストを分割することで同様に並列化を行っている.

4 章で SSFA を用いた並列マッチングの計算量は入力長 n, 正規表現の長さ $|r|$, 並列度 p について $O(n/p+p|Q_N|^2)$ であることを示した. 図 6 から 6 並列までは O0, O3 ともにスケールしていることがわかる. O0 は 12 並列までスケールしいるのに対し, O3 及び Read は 7 並列で極端に性能が下がっているが, これは Intel の Hyper-Threading [7] が物理 6 コアの各コアに対して 2 スレッド分の命令をスケジューリングすることで仮想 12 コアの並列実行を行っているからである. O3 マッチングではコード生成によって最適化された x64 ネイティブコードが実行され, 1 スレッドで 1 コアのリソースを使い尽くしているものと思われる. Read では最大スループットが 20GB/sec 近く, 文字列のメモリー読み出し速度の限界によって O3 並列マッチングのボトルネックとなったわけではないことがわかる.

結果としてマッチングを並列化することによって, データ主導のマッチング (O0) では仮想 12 コアに対して 12 倍, 4 章説明したでコード生成+最適化を行ったマッチング (O3) では物理 6 コアに対して 6 倍の性能を出すことができた.

6.3.2　小さなテキストに対するベンチマーク

比較的規模の大きな入力文字列に対しては, 並列実行の台数効果によってマッチングの並列化を行うほうが高速に実行できることは図 6, 表 6 より明らかである. それでは比較的規模の小さな入力文字列に対して, スレッドの生成やスレッドごとの集計処理などのオーバーヘッドがどこまでマッチング全体の遅延となるのだろうか. これらを調べるため 100~1000KB の入力に対して O0,O3 それぞれ通常のマッチングと最もオーバーヘッドの低い並列度である 2 並列マッチ

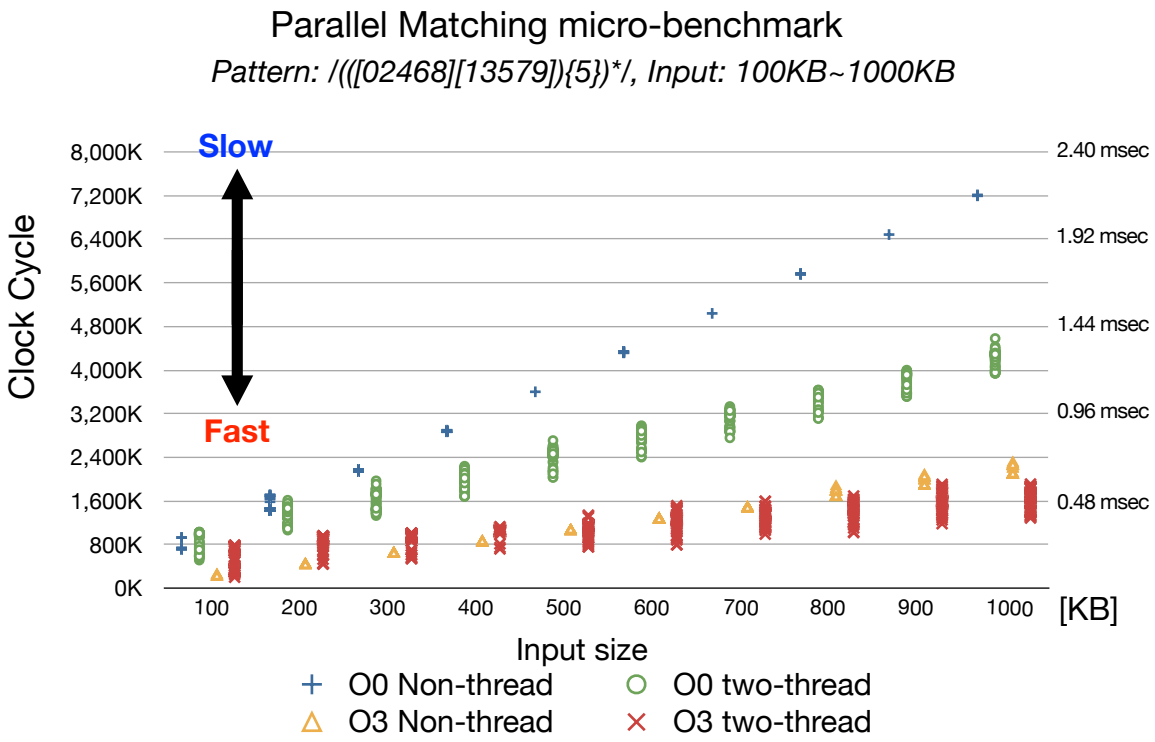

図 7　小さなテキストに対する並列マッチング，それぞれ 100 回の実行時間をプロットしている．この時 DFA の状態数は 10 個，SSFA の状態数は 21 個．

ングを行なった．スレッドのスケジューリングによるばらつきを考慮しそれぞれ初回実行を除いて 100 回実行した結果を図 7 にプロットしている．実行時間は rdtsc 命令によって計測したクロック数で，クロック周波数は 3.33GHz なので 1 クロックサイクルは $(3.33\times10^{9})^{-1}\approx0.3\times10^{-9}[sec]=0.3[nsec]$ となる．

図 7 において，2 スレッドによる並列マッチングは O0,O3 いずれも数百万クロック程度のばらつきがありこれは数 100KB 程度のテキストだと大きな遅延となる．しかし結果的には O0 では 300KB，O3 では 800KB 付近で 2 スレッドの並列マッチングが安定して速度が上回っていることが読み取れる．

7　関連研究

正規表現からコード生成を行う研究として Thompson [9] は NFA ベースのバックトラックを行わないコードを IBM 7094 の機械語として生成する手法を提案しており，この NFA ベースのマッチング手法は Thompson NFA と呼ばれている [2]．

DFA の状態数爆発問題について，Fang ら [6] は最左最短かつ重なりのないマッチング規則 (*non-overlapping left-most shortest match*) に基づいた正規表現の書き換え規則を提案した．不正アクセス監視システム/侵入検知システム (IDS) の Snort [18] のルールセットに含まれる 222 の正規表現，パケット解析ツール l7-filter [10] で使用されている 70 の正規表現について DFA の状態数を十分に処理可能な量に抑えることで実システムでの DFA ベースマッチングの有用性/汎用性を証明している．

また，並列化マッチングの研究としては松崎ら [13] は NFA の状態数 $|Q_N|$，入力文字列の長さ n，並列度 p において $O((n/p+\log p)|Q_N|^3)$ の NFA ベースの並列マッチング，同様に DFA の状態数 $|Q_D|$ において $O((n/p+\log p)|Q_D|)$ の DFA ベースの並列マッチングを実装し，Hadoop 上で並列マッチングについての検証を行なっている．これに対して，本研究では SSFA を

用いることで NFA ベースで $O(n/p+p|Q_N|^2)$, DFA ベースで $O(n/p+p)$ の並列マッチングを実現している.

オートマトンには様々な拡張モデルが存在する. その中でも並列性を取り入れたモデルとして, 並列オートマトン (*Parallel FA*) [20], 並行オートマトン (*Concurrent FA*) [21], 交代性オートマトン (*Alternation FA*) [15] などがあるが, これらは並列性を持つモデルを扱うための拡張でありオートマトンそのものを並列実行する拡張ではない. 本研究で提案した SSFA とこれらのオートマトンに関連はない. SSFA 単体では受理/非受理を行うことはできないので厳密には順序機械と言うべきかもしれないが, 並列マッチングにおいて DFA 的に状態遷移を行い最終的に受理/非受理を判定できるため本研究で SSFA と命名した.

8 まとめと今後の課題

正規表現から DFA を構築し, さらに最適化されたコードを動的に生成する手法で, 従来のデータ主導の DFA マッチングを正規表現に依らず 3 倍, 正規表現によっては 6 倍高速化することに成功した. さらに SSFA を用いてマッチングを並列化することで台数分の並列効果が得られることを示し, データ主導マッチングで 12 倍コード, 生成マッチングで 6 倍と物理/仮想コアに対して台数効果を出した. 最終的に既存のデータ主導の非並列マッチングと比べ, 並列化とコード生成を併用することで 30 倍の速度向上を実現し 14GB/sec に近いスループットを出すことに成功した. 本研究によって実装した正規表現エンジンはソースコードを公開している [17] ので, 誰でも自由に使うことができ本論文中の実験を再検証可能である. †4

今後の課題として, SSFA の定性的な平均状態数の解析や状態数が $|Q_S| = O(2^{|Q_N|^2}), |Q_S| = O(|Q_D|^{|Q_D|})$ となる正規表現についての考察, マッチした文字列を記憶しておく **submatch** [19] の実装とコード生成/並列化への応用が挙げられる.

†4 エンジンはまだ研究開発段階であり, 現時点でドキュメントや基本 API の整理は行なってない. また, コード生成は X64 アーキテクチャのみに対応している.

謝辞 正規表現マッチングが並列実行可能であることを教えて頂いた河野真治氏 (琉球大学), 並列マッチングについて議論して頂いた松崎公紀氏 (高知工科大学), 本実装について全面的に開発支援を行ってくれたサイボウズ・ラボユース及びサイボウズ・ラボのメンバー, 特に竹迫良範氏と西尾泰和氏, 蓑輪太郎氏, 中谷秀洋氏に感謝する.

参 考 文 献

[1] Aho, A. Sethi, R. Ullman, J : Compilers: Principles, Techniques, and Tools Second Edition (2006). pp.147–166.
[2] Cox, R : Regular Expression Matching Can Be Simple And Fast. (2007) Available at: `http://swtch.com/~rsc/regexp/regexp1.html`
[3] Cox, R : Regular Expression Matching: the Virtual Machine Approach. (2009) Available at: `http://swtch.com/~rsc/regexp/regexp2.html`
[4] Cox, R : Regular Expression Matching in the Wild. (2010) Available at: `http://swtch.com/~rsc/regexp/regexp3.html`
[5] Cox, R : re2 - an efficient, principled regular expression library. Available at: `http://code.google.com/p/re2/`
[6] Fang ,Y. Zhifeng ,C. Yanlei, D.: Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection. *ACM/IEEE symposium on Architecture for Networking and Communications Systems(2006)*. pp. 93-102.
[7] Intel 64 and IA-32 Architectures Software Developer's Manuals. Available at `http://www.intel.com/products/processor/manuals/`
[8] Jiang, J. Wang, X. He, K. Liu, B. : Parallel Architecture for High Throughput DFA-Based Deep Packet Inspection. *Communications (ICC), IEEE International Conference(2010)*. pp. 1–5.
[9] Thompson, K : Regular Expression Search Algorithm. *Communications of the ACM 11(6) (June 1968)*. pp. 419—-422.
[10] l7-filter — ClearFoundation. Available at `http://l7-filter.clearfoundation.com/`
[11] McNaughton, R. Yamada, H : Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers EC-9(1) (1960)*. pp. 39–47.
[12] Michael Sipser : 計算理論の基礎 第二版 1 オートマトンと言語. pp. 36–66.
[13] 松崎公紀, 胡 振江, 武市正人 : 正規表現正規表現マッチングとその Hadoop 上での評価. 情報処理学会 第 83 回プログラミング研究発表会 (2011)
[14] 松坂 和夫 : 集合・位相入門. pp. 1–39.
[15] 守屋 悦朗 : 形式言語とオートマトン. pp. 102–105.
[16] 新屋 良磨, 河野 真治 : 動的なコード生成を用いた正規表現評価機の実装. 第 52 回プログラミング・シン

ポジウム (2011)

[17] 新屋 良磨 : Regen - Regular Expression Generator,Compiler,Engine. Available at : `https://github.com/sinya8282/regen`

[18] Snort :: Home Page. Aailable at `http://www.snort.org/`

[19] Laurikari, V : NFAs with Tagged Transitions, their Conversion to Deterministic Automata and Application to Regular Expressions. *Proceedings of the Symposium on String Processing and Information Retrieval, September(2000)*. pp. 181–187.

[20] Stotts, D, P. Pugh, W. : Parallel Finite Automata for Modeling Concurrent Software Systems. *Journal of Systems and Software, vol 27(1994)*. pp. 27–43.

[21] Zetzsche, G. Jantzen, M. Manfred, K. : Concurrent Finite Automata. *Tagungsband 17. Theorietag Automaten und Formale Sprachen (2007)*. pp. 84–88.

[22] 光成 滋生 : Xbyak - x86, x64 JIT assembler. Available at `http://homepage1.nifty.com/herumi/soft/xbyak.html`