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 †

我々は実行時コード生成や並列化を用いた高速な正規表現エンジンを実装し, 外部プログラムから利用可能なライブラリとして Regen を開発している. 正規表現エンジン・ライブラリは数多くの既存実装があるが, それらは実装方法や性能が大きく異なる. 本論文では Regen における設計方針や特長を紹介し, 既存実装との違いを説明する. また, 既存実装の中で特に高速な Google RE2 との比較を元に性能に関する考察を行なった.

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

RYOMA SHINYA ,† SHIGEO MITSUNARI ††
and MASATAKA SASSA†

In this paper, we introduce a fast regular expression library Regen. Our library uses dynamic code generation and parallelization to achieve efficient pattern matching. We describe of the design of Regen, and compare with proposed methods of existing implementations. Furthermore, we discuss the performance with Google RE2.

1. はじめに

正規表現はテキスト処理やパケットフィルタリング等多くの場で活用されており, 機能を提供する正規表現ライブラリの性能は重要である. 現在数多くの正規表現ライブラリやプログラミング言語組み込みの正規表現エンジンの実装があるが, それらは実装方法や性能が大きく異なっており長所/短所を併せ持っている. 一方我々の開発している Regen[2] はマッチング速度に特化したオープンソースな正規表現ライブラリであり, 既存実装に無い高速化手法として動的なコード生成と並列マッチングを実装している[1]. 本論文ではまず正規表現とエンジンの実装方法について説明し, Regen の設計方針と特長を紹介する. さらに, 既存実装の中で特に高速な Google RE2[5] との比較を元にした考察を行った. 最後に課題や今後の方針をまとめる.

本論文は日本ソフトウェア科学大会で発表した手法[1] を実装した正規表現ライブラリ Regen の解説論文である.

† 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
†† サイボウズ・ラボ株式会社
Cybozu Labs, Inc

2. 正規表現

正規表現は形式言語理論で導入された文法で, 計算理論におけるオートマトンと深く関連している. 本論文及び Regen で対象とする正規表現の定義は等価な有限オートマトンが構成可能なものを指し, 基本的に以下の文法によって記述される.

$$
\begin{array}{rll}
\text{正規表現 } r = & \epsilon & \text{空文字} \\
| & c & \text{文字} \\
| & rr & \text{連接} \\
| & r|r & \text{集合和} \\
| & r* & \text{閉包}
\end{array}
$$

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[3),4)]. 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）することを繰り返す. 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」, 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.

表 1 実装方法の比較

正規表現サイズ: m, 文字列長: n	DFA	バックトラック NFA	Thompson-NFA
コンパイル時最悪時間（空間）計算量	$O(2^m)$	$O(m)$	O(m)
マッチング時最悪時間計算量	$O(n)$	$O(2^n)$	$O(mn)$
マッチング時最悪空間計算量	$O(1)$	$O(n)$	$O(m)$
拡張機能の実装し易さ	×	◎	○

2.1 拡張された正規表現

正規表現は古くから Unix 系コマンド等で利用されており, 多くの拡張機能を取り入れた正規表現が広まっている. 形式言語理論における正規表現の能力を超えない拡張記法（糖衣構文）として

拡張記法	意味
$r?$	$r\|\epsilon$
$r+$	$rr*$
$r\{m,n\}$	m 回以上, n 回以下の r の繰り返し
.	全ての文字の集合和
$[c_1c_2]$	$c_1\|c_2$

等が広く使われており, これらの記法は Regen でも利用可能である.

しかし, 正規表現本来では未定義（あるいは不可能）である, より根本的な拡張機能も広く普及している. たとえば, 正規表現に部分的にマッチした文字列位置を保持するキャプチャ(capture, submatch) や文字列を正規表現内部で参照可能な後方参照などの機能である. これら拡張機能については Regen の対象から外れており, これらの実装方法や複雑さの説明は行わない.

3. 正規表現エンジンの実装

正規表現エンジンの実装には, 有限オートマトンが用いられる. 有限オートマトンは大きく DFA/NFA に分けられ, さらに NFA では深さ優先探索型（バックトラック NFA）の実装と幅優先探索型の実装 (Thompson-NFA) がある[3),4)]. これらは正規表現から有限オートマトンの構成, 及び文字列に対してのマッチングにおいて時間（空間）計算量が大きく異なる. 表 1 に特徴をまとめた.

既存実装の多く☆はバックトラック NFA を採用[4)] しており, これらは拡張機能の対応や正規表現からのコンパイル時計算量を優先したものと思われる. 一方, マッチング時の計算量が最も高速な DFA 実装として Google RE2 が 2010 年に公開されており, 高速かつ固定メモリ消費マッチングを行えスレッドセーフな優れた実装となっている.

☆ PCRE, 鬼車, Java や各種スクリプト言語組み込み

4. Regen の設計方針と特長

ここでは実装の詳細については立ち入らず, Regen の特長と設計方針について説明する.

Regen はマッチング時の性能を追求した正規表現ライブラリであり, 基本的に DFA を用いてマッチングを行う. 既存実装とは全く異なる高速化手法として動的なコード生成, さらには並列マッチングを実装している[1)]. 本節では以下に示す Regen の特長を説明していく.

- 動的なコード生成
- 並列マッチング
- マッチした文字列全体の特定
- 最長/最短マッチのサポート
- 欲張り/控えめマッチのサポート
- 否定, 集合積, 限定再帰のサポート

4.1 マッチングの高速化

Regen では高速化のために内部で DFA を基にした動的なコード生成を行なっている. コード生成には JIT ライブラリである Xbyak[10)] を用いており, x86/x64 マシンを対象に開発している. さらに文字列長 n, p 並列で計算量が $O(n/p+p)$ の効率の良い並列マッチングも実装しており, ライブラリのユーザーはオプションを指定して並列マッチングを行うことができる. コード生成と並列化の実装の詳細は, 我々の既存研究[1)] を参照してほしい.

4.2 マッチ文字列の特定

正規表現を用いた多くのテキスト処理には, マッチした文字列の位置を取得する機能が必要とされている. Regen ではマッチした文字列 “全体” の位置を取得する機能をサポートしているが, 部分的な文字列の取得, いわゆるキャプチャ(capture, submatch) はサポートしてない. 例えば正規表現/(a|b)*(a|c)*/で文字列 “zzabaczz” を部分マッチ検索した場合, 全体として “abac” がマッチする. さらに正規表現の最初の括弧に囲まれた/(a|b)*/には “aba” が, 後ろの括弧に囲まれた/(a|c)*/には “c” がマッチすると解釈でき, キャプチャはこの部分文字列も特定することができる. Regen でキャプチャをサポートしない理由は, キャプチャの効率の良い DFA エンジンの実装が困難[8)] だか

らであり, Google RE2 もキャプチャが必要な場合は内部的に NFA を用いている.

キャプチャには対応していないが, 全体の文字列は取得できるということで最長/最短マッチ, さらには欲張り/控えめ (greedy/non-greedy) なマッチ動作の対応している点は Regen の大きな特長である.

4.3 拡張演算

Regen では拡張演算として否定, 集合積, 限定再帰を実装している. 否定/集合積は形式言語理論でいう拡張正規表現の演算[7] だが, 限定再帰は Regen 特有の演算である. ここでは限定再帰のみ説明を行う. 限定再帰演算は通常の正規表現の能力を超える再帰的な文法を制限した文法で, 具体的には再帰の深さに上限を設ける. たとえば 1 段から 3 段までネストした括弧は限定再帰演算子 "@" を用いることで/⟨@{0,3}⟩/と記述でき, これは/⟨(⟨(⟨(⟨⟩)?⟩)?⟩)?⟩/と等価である.

否定, 集合積, 限定再帰を用いて複雑な正規表現をより簡潔に記述することが目標であり, しかもこれらの演算はマッチング速度に影響を及ぼさない（但し, 正規表現のサイズは大きくなる場合が多い）.

4.4 DFA の状態爆発について

正規表現から等価な DFA の構成について状態爆発問題がある. 例えば正規表現 $(a|b) * a(a|b)\{9\}$ は等価な最小 DFA の状態数が $2^{9+1} = 1024$ 個となり, 正規表現のサイズに対して指数関数的に増加する. これを DFA の状態爆発問題 (*State explosion problem*) という.

状態数爆発を起こすような正規表現は比較的限られたパターンであり, 典型的な正規表現ではサイズ m に対し最小 DFA の状態数が $O(m^3)$ 程度に収まることが知られている[3]. しかし上記のような状態爆発を起こす正規表現が与えられ, コンパイルに時間がかかってマッチングがいつまでも行えないエンジンでは汎用的に使いづらい. これは全ての DFA ベースの正規表現エンジンの課題である.

状態爆発問題への対策は大きく分けて 3 つある.

(1) DFA 構築の遅延評価する
(2) 正規表現を書き換える[6]
(3) 部分的に DFA を構成する[9]

(1) は正規表現から NFA を構成し, 最初は対象文字列に対して幅優先 (Thompson-NFA) でマッチングを行う. 内部では文字列から 1 文字読み込む度に状態の集合を更新している. この状態の集合と文字による遷移関係を保存しておくことで, 次に同じ状態の集合と文字を読み込んだ時の計算を省略することができる. これは最初で DFA 全体を構築せず, 文字列を読み込みながら部分的に構築行なうこととなり, マッチング時の最悪計算量は Thompson-NFA と同等で $O(mn)$, 最良計算量は DFA と同等で $\Omega(n)$ となる. この技法は On-the-Fly DFA Construction とも呼ばれ, Regen や Google RE2 で実装されている.

(2), (3) については本実装で直接対応してはいない. 詳細は参照論文を参照してほしい.

5. 性能考察

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った. 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 命令によって取得したクロック数を用い, 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/Hyper-Threading を無効にした Intel Core i7-980X (3.33GHz, 6 物理コア), 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エンジンを以下に示す.

Regen 本実装. 内部でコード生成を行わない.
Regen JIT 本実装. 内部でコード生成を行なう.
RE2 Goole RE2
Read 文字列を読むだけの指標プログラム.

5.1 マッチング速度の比較

まず, Regen における動的なコード生成を用いたマッチングの高速化を計測する. ベンチマークでは単純な正規表現を用いているが, DFA では正規表現の複雑さはマッチング時の時間計算量に関係しない. 結果を表 2,3 に載せる.

ここでは Regen が内部でコード生成する際に, 最適化が良く効くパターンと全く効かないパターンの 2 種類の正規表現を用いた. 表 2, 2 から Regen においてコード生成を用いたマッチングでは 3 倍, さらに最適化が効く場合は 6 倍高速な結果が出ていることが分かる. 既存実装で高速な Google RE2 に対して 6 倍以上, スループットにして 2.7GB/sec 以上の結果から,

正規表現:/(0123456789)*/, 入力:1GB

Engine	Codegen	Matching	Throughput
RE2	54668	12630176840	0.263GB/sec
Regen	190184	7414917052	**0.449GB**/sec
Regen JIT	423684	1225896300	**2.716GB**/sec
Read		551121603	6.042GB/sec

表 2 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
Regen	236960	7417497860	0.448GB/sec
Regen JIT	418368	2188205292	**1.521GB**/sec
Read		551121603	6.042GB/sec

表 3 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

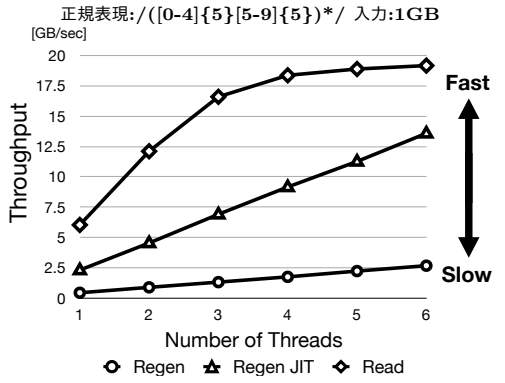

図 1 大きなテキストに対する並列マッチング. この時の DFA の状態数は 10 個, 並列化 DFA の状態数は 109 個.

コード生成の高速化の恩恵は非常に大きい.

5.2 並列マッチングの効果

次に Regen における並列マッチングによる効果を計測する. ここでは並列マッチングを実装していない Google RE2 は比較に用いない. 1GB のテキストに対して,1~6 スレッドで並列マッチングを行なった結果を表 1 に示す. 結果から, スレッド数に対して台数効果が確認することができ, コード生成を用いた場合 6 スレッドでは 13GB/sec 以上の性能が出ている.

ここでは比較的単純な正規表現を用いてベンチマークを行なった. 正規表現と対象文字列の組み合わせ, 実行環境のキャッシュサイズ等の実験環境によってこれらの結果は異なるかもしれない. Regen はライブラリとして公開[2] しており, 各自の条件でベンチマークを行うことができる.

6. ま と め

実効性能に特化した正規表現ライブラリ Regen の設計方針や特長を紹介し, Google RE2 との比較を用いることで Regen の性能について議論した. 現在も Regen は改良や機能拡張のため実装を行なっており, ライブラリとしてはまだ開発段階である. 本論文内で紹介した機能やベンチマークについては, 是非 Regen を使用して各自の環境で実行してもらいたい.

謝 辞

Regen において全面的に開発を支援してくれたサイボウズ・ラボ及びサイボウズ・ラボユースのメンバーに感謝する.

参 考 文 献

1) 新屋 良磨, 光成 滋生, 佐々 政孝 : 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. 日本ソフトウェア科学会第 28 回大会 (2011)
2) 新屋 良磨 : Regen - Regular Expression Generator,Compiler,Engine. Available at : `https://github.com/sinya8282/regen`
3) Aho, A. Sethi, R. Ullman, J : Compilers: Principles, Techniques, and Tools Second Edition. pp.147–166 (2006).
4) Cox, R : Regular Expression Matching Can Be Simple And Fast. (2007) Available at: `http://swtch.com/~rsc/regexp/regexp1.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*. pp. 93-102 (2006).
7) 田中 哲: 正規表現における非包含オペレータの提案. プログラミングシンポジウム報告書 Vol49th pp. 55-64 (2008).
8) 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*. pp. 181–187 (2000).
9) Yang, Y.-H.E. Prasanna, V.K. : Space-time tradeoff in regular expression matching with semi-deterministic finite automata. *INFOCOM, 2011 Proceedings IEEE* . pp. 1853–1861 (2011).
10) 光成 滋生 : Xbyak - x86, x64 JIT assembler. Available at `http://homepage1.nifty.com/herumi/soft/xbyak.html`