PaREM: A Novel Approach for Parallel Regular Expression Matching

(CSE-2014, ©IEEE)

Suejb Memeti and Sabri Pllana
Department of Computer Science, Linnaeus University
351 95 Växjö, Sweden
{suejb.memeti, sabri.pllana}@lnu.se

***Abstract*—Regular expression matching is essential for many applications, such as finding patterns in text, exploring substrings in large DNA sequences, or lexical analysis. However, sequential regular expression matching may be time-prohibitive for large problem sizes. In this paper, we describe a novel algorithm for parallel regular expression matching via deterministic finite automata. Furthermore, we present our tool PaREM that accepts regular expressions and finite automata as input and automatically generates the corresponding code for our algorithm that is amenable for parallel execution on shared-memory systems. We evaluate our parallel algorithm empirically by comparing it with a commonly used algorithm for sequential regular expression matching. Experiments on a dual-socket shared-memory system with 24 physical cores show speed-ups of up to $21\times$ for 48 threads.**

***Index Terms*—parallel processing, multi-core, regular expression, finite automata**

I. Introduction

There are many relevant applications of regular expression matching (REM) and finite automata (FA) including DNA sequence matching [1], network intrusion detection [2], and information extraction from web based documents [3]. The computational complexity of pattern finding grows with increasing the number of states of the automaton and the size of the input. While the stagnation in processor clock rates promises no performance increases for sequential implementations of REM, availability of affordable multicore processors provides opportunities for significant improvement. For instance, the recently introduced Intel® Xeon® Processor E7-8890 v2 manufactured at 22nm comprises 15 physical cores and supports 30 threads or so called logical cores. Shared-memory systems with up to eight processors of this type are feasible that would lead to a system with 240 logical cores. To exploit these powerful systems, scalable parallel REM implementations are required.

Programming and resolving problems within automata theory is a relatively complex and time-consuming process, and still the results may not be reliable because of the chances to have an incorrect FA representation. Furthermore, efficient parallel programming of multicore systems is complex and this issue is known in the literature as the *"programmability wall"* [4]. Democratization of parallel REM would benefit from tools that hide parallel programming from the end-user and automatically generate the correct parallel implementation that is ready for compilation and efficient execution.

Various approaches for increasing the performance of REM evaluation have been proposed. For instance, Maine [5] is a library for data-parallel FA, which formalizes the evaluation of a FA as a matrix multiplication. Holub and Stekr [6] propose an algorithm for parallel execution of synchronized deterministic finite automata (DFA). Yang and Prassana [7] introduce an approach that uses segmentation for regular expression evaluation via nondeterministic finite automata (NFA). In [8] authors propose the range-coalesced representation of transition table to optimize the cost of the transition table lookup for each active state. While there are model to text generators (such as, Acceleo [9]), or RE to NFA-DFA converters (such as, JFLAP [10]), to our best knowledge there are no automatic parallel code generators for RE or FA.

In this paper, we describe a novel algorithm for Parallel Regular Expression Matching (PaREM) that scales gracefully for various problem sizes and number of threads. The algorithm was devised to be efficient for general automata independently from the number of states, and for large spectrum of input text-sizes. Our algorithm is optimized to do very accurate speculations on the possible initial states for each of the sub inputs (split among the available processing units), instead of calculating the possible routes considering each state of the automaton as initial state. This method is more effective when the adjacency matrix (used for graph representation of the automaton) is sparse, although it shows major improvements in dense matrices as well. To ease the access to the proposed parallel algorithm for a broad spectrum of users (including the users without background in parallel programming), we have developed our tool PaREM that can transform automatically a Regular Expression (RE) or FA into the corresponding code (C++ and OpenMP) for our algorithm that is amenable for parallel execution on shared-memory systems. Experimental results on a dual-socket shared-memory system with 24 physical cores show a close to linear speedup compared to the sequential implementation for problem sizes comparable to the cache size and significant speedup for larger problem sizes that use further levels of memory hierarchy.

The main contributions of this paper include:

- A scalable algorithm for parallel regular expression matching;
- PaREM tool that automatically generates parallel code

from a given regular expression or finite automata;
- Empirical evaluation of the proposed parallel algorithm and the PaREM tool using a modern dual-socket shared-memory system with 24 physical cores.

The rest of the paper is organized as follows. Section II provides background information on regular expressions and finite automata and presents our parallel algorithm. Section III describes the implementation of the PaREM tool, and Section IV the corresponding experimental evaluation. The work described in this paper is compared and contrasted to the related work in Section V. Section VI provides a summary of our work and a description of future work.

II. Methodology

A. Background

A regular expression is a string for describing search patterns. A finite automaton is a graph-based way for specifying patterns [11]. Finite automata and regular expressions may be used in pattern finding algorithms.

Deterministic Finite Automata (DFA) is a quintuple of $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is set of symbols (alphabet), $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, q_0 is the initial state and F is the set of final states [11] [12]. A DFA operates in the following manner: when a program starts, the current state is assumed to be the initial state q_0, on each character the current symbol is supposed to move to another state (including itself). When the input reaches the last character, the string is accepted if and only if the current state is in the set of final states. It is called deterministic because in each state and for each input symbol a unique transition is defined.

Nondeterministic Finite Automata (NFA) is defined by the quintuple $(Q, \Sigma, \delta, q_0, F)$ as in DFA except the alphabet may contain an empty symbol; the transition function returns a set of states rather than a single state. It is called non-deterministic because of the choice of moves that may lead from one state to another.

B. Parallel REM Algorithm

Existing approaches for parallel REM (such as [6]) split the input into smaller substrings among all or a selected number of processing units, run the automaton on each of them, and join the sub- results. While other approaches calculate the possible initial states from each state of the automaton, our algorithm takes a step ahead by excluding all the states that the automaton has no outgoing or incoming transitions for the specified characters. Calculating the possible routes from each state of the automaton becomes time-consuming and memory-expensive for large finite automatons.

The basic idea of the sequential REM or DFA is that one starts from q_0 and after n (input length) steps another state from set Q is reached. Its time complexity depends only on the input length.

Our algorithm is based on domain decomposition, which means it slices the input in p parts (see Algorithm 1), where p is the number of processing units (line 3). For each p_i the possible initial states R are determined by finding the intersection of possible initial states $R = S \cap L$ (line 5 — 15). S is the set of initial states for the first character of T_{p_i} (that is, the sliced input for this specific processor) where $q_i \in S$ if $\exists : \delta(q_i, T_{p_i})$. L is the set of initial states for last character of T_{p-1_i}, where $L_i = \delta(q_i, T_{p-1_i})$. Each chunk of the input is mapped to a processing unit, and each processing unit is responsible for finding the possible initial states for its own chunk of the input. The processing unit with $ID = 0$ already knows the possible initial state, that is q_0, so a calculation for determining the possible initial states is not necessary. For each state in R, a REM is done and the result is stored in I (lines 16 — 25).

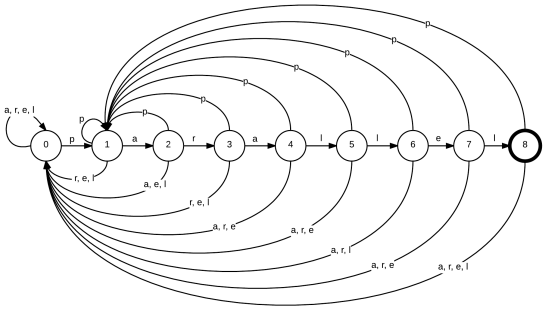


Fig. 1: Automaton A for matching the pattern *parallel*

When all processors have finished their jobs, a binary reduction of the final results is completed. The reduction is done by connecting the last active state of P_i to the first active state of P_{i-1}. The connection is accepted only if a transition from last active state of P_i to the first active state of P_{i+1} exists with the first character of the sub-result of next processor T_{p+1},$\delta(q_i, T_{p+1_i})$. An input is accepted only if for each processor there exist a sub route, which can be connected with the result of the previous and next processor's result, and the last state of the automaton is member of the final state set. The worst-case scenario would be if all the states have the same input and output transitions.

C. Description of PaREM Algorithm with an Example

To show how the possible initial states are determined, the following example from Fig. 1 is used. Let T be an input string, $T =$ "$plaraparallelapareparapl$" and assume that we will use four processing units (that is threads).

The transition table corresponding to the automaton from Fig. 1 is shown on Table I. The transition table for this automaton is dense, which will produce a dense adjacency matrix.

The input length is 24 characters, so when split among processing units we get four substrings of six characters ($P_0 =$ "$plarap$", $P_1 =$ "$aralle$", $P_2 =$ "$lapare$" and $P_3 =$ "$parapl$"). Table II shows the accurate possible initial states found for each of the processor's input, and the visited states starting from each of the possible initial states. In this example, each state has exactly the same amount of outgoing

Algorithm 1 Parallel Regular Expression Matching (PaREM)

%Input: Transition table Tt, set of final states F, input T%
%Output: Result of REM%
1: $I = vector(p)$ /* initialize final result vector */
%$P_0 ... P_p ...$ processing unit, p is the total number of processing units %
2: **for** $P_0, P_1, ..., P_p$ **do** in parallel
3: $start\_position = i * (T.length/p)$
4: $pi\_input = substring(start\_position, T.length/p)$
%start find possible initial states %
5: **for** $q_0, q_1, ..., q_n$ **do**
% $pi\_input.at(0)$ returns the first char of pi_input %
6: **if** $(Tt[q_i][pi\_input.at(0)] \in Q)$ **then**
7: $S[i] = q_i$
8: **end if**
9: **end for**
10: **for** $q_0, q_1, ..., q_n$ **do**
% $pi\_input.back()$ returns the last char of pi_input %
11: **if** $(Tt[q_i][pi\_input.back()] \in Q)$ **then**
12: $L[i] = Tt[q_i][pi\_input.back()]$
13: **end if** se le da una columna [][pi_input.back()]
14: **end for**
%end find possible initial states %
15: $R = S \cap L$ %intersection of possible initial and last states %
16: **for** $r \in R$ **do**
17: $Rr = vector(pi\_input.length())$
18: **for** $char \in pi\_input$ **do**
19: **if** $(Tt[r][char] \in F)$ **then**
20: $found++$
21: **end if**
22: $Rr[i] = r = Tt[r][char]$
23: **end for**
24: $I[i].push\_back(Rr)$
25: **end for**
26: **end for**
% Wait for the slowest processor%
% Perform a reduction of I%

TABLE I: Transition table for automaton on Fig. 1

δ_A	p	a	r	e	l
0	1	0	0	0	0
1	1	2	0	0	0
2	1	0	3	0	0
3	1	4	0	0	0
4	1	0	0	0	5
5	1	0	0	0	6
6	1	0	0	7	0
7	1	0	0	0	8
8	1	0	0	0	0

transitions, which means there is a transition from each state for each symbol of the alphabet.

The set of DFA initial states R is equal to the set of states

TABLE II: Possible initial states for P_0, P_1, P_2 *and* P_3

	$S \cap L$	Visited states
P_0	0	1 0 0 0 0 1
P_1	1	2 3 4 5 6 7
P_2	0	0 0 1 2 3 0
	7	8 0 1 2 3 0
P_4	0	1 2 3 4 1 0

L achieved from the last character of the input string of the previous processor, because S is equal to set of all states. Therefore, $R = S \cap L = L$. This applies only to dense transition tables, because from each state on any symbol is possible to go to another state (including itself). In practice, most of DFA produce a sparse transition table. In sparse transition tables the set of states S achieved from the first character of the input string that is mapped to the processing unit, is determined by the outgoing transitions of states for a specific character. We treat each matrix as sparse, that is why $R = S \cap L$. It is possible to identify a sparse matrix, but inspecting each element of large matrices whether is empty or not may be time-consuming.

The highlighted numbers on Table I represent the set of states S and L for P_2, where S (colored in green) is set of source states for which a transition exist on "l" (first character of the input mapped to P_2), and L (colored in yellow) is set of unique destination states for which a transition exists on "e" (last character of the input string mapped to P_1).

The general enumeration approach of REM algorithms calculates possible routes (moving from one state to another) considering each state of the automaton as initial state. In this example, the enumeration approach of REM would have performed $3 \times 9 + 1$ (three processing units (P_1, P_2 and P_3) would start from all the nine possible states, and P_0 would start from state q_0) calculations. Our algorithm performs only five calculations for this example, and we believe that this number becomes lower for sparse transition tables. If the input of processing unit P_{i-1} would end with "l", there would be four (0, 5, 6, 8) possible initial states. The worst-case scenario would be if each of the sub-inputs ends with "l"; in such case $3 \times 4 + 1$ calculations are performed for dense matrices that is an improvement by $2.15 = (3 * 9 + 1)/(3 * 4 + 1)$, compared to the general approach.

III. Implementation

Fig. 2 depicts our PaREM tool, which takes as input a RE or a FA and generates the corresponding C++ code representation of the given RE or FA. The generated C++ code includes OpenMP [13] directives and routines and is in accordance with our Algorithm 1. In the process of PaREM implementation, we have specified a context-free grammar to define the language that accepts regular expressions as input. Table III lists the accepted operators by PaREM context-free language.

The *Klenee Star* denotes zero or more occurrences of a symbol or sub-expression (for instance, ϕ, a, aa, aaa, where ϕ is an empty transition). The NFA representation of the

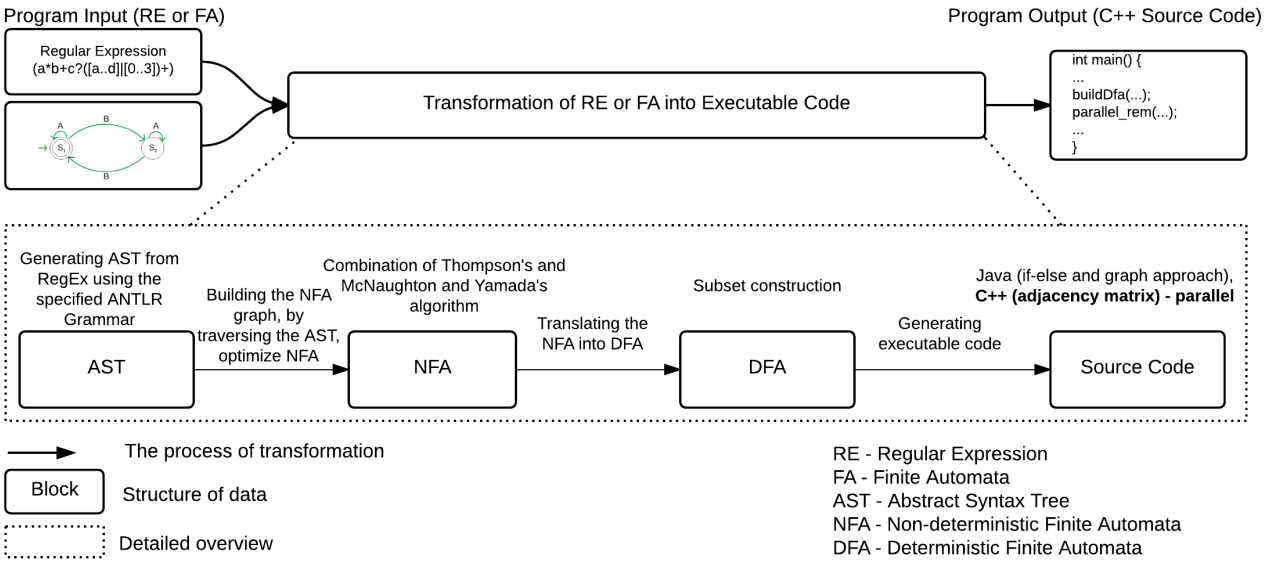


Fig. 2: The use of PaREM tool for translating regular expressions into equivalent finite automata (NFA, then DFA) and generating source code (C++ and OpenMP) that represent the same given RE or FA

Klenee Star is shown in Fig. 4d. The *Positive Closure* also known as *Repetition* is an extended operator of the *Klenee Star*, which denotes one or more occurrences of a symbol or sub-expression (for instance, $a+$, Fig. 4f, is equal to $aa*$ that results to these possibilities: $a, aa, aaa,)$.

The *Union* operator (represented as NFA in Fig. 4c), expressed by a vertical bar, provides the possibility to choose between two or more sub-expressions (such as, a, b). The *Range* (defined based on ASCII code order) operator, or *Character Class*, is an extended operator of *Union*, instead of writing $0|1|2|3$ the *Range* operator $[0..3]$ can be used. It applies to integers and characters.

The *Optionality* operator (shown as NFA in Fig. 4e) denotes zero or one occurrence of a symbol or sub-expression (for instance, $a? = \phi|a$). The *Group* operator is introduced to change the operator precedences. For instance, $a|b*$ and $(a|b)*$ produce different results, in the first example the *Klenee Star* operator has priority over the *Union* operator, while in the second example the *Union* operator has a higher priority. By combining these operations (using *Concatenation* operator, Fig. 4b) arbitrarily complex regular expressions can be written.

For each RE a specific Abstract Syntax Tree (AST) is generated that represents the abstract syntactic structure of the RE. For easier translation into a target structure, additional details have been added (such as, the node type) to the AST. The generated AST can have an arbitrary number of sub-trees, which in essence are ASTs [14]. Fig. 3 shows an example of how an AST is constructed for a given RE. Dashed-line compartments indicate the sub-trees.

The priority of the *Union* operator over the *Quantifier* operator in the sub-expression "$(a|b)?$" is depicted in Fig. 3. The deeper the operator is in the AST hierarchy, the higher priority it has.

TABLE III: PaREM's Accepted Regular Expressions Operators

Operator	Name	Description
ab	Concatenation	b right after a
$a*$	Klenee Star	zero or more a's
$a\|b$	Union	either a or b
$a+$	Positive closure	one or more a's
$[0..9]$	Range	either $0, 1...$ or 9
$a?$	Optionality	zero or one a
$(ab\|c)*$	Group	zero or more of either ab's or c's

We transform the AST into NFA graph using the *McNaughton-Yamada-Thompson Algorithm*. To preserve the operator priority, the depth-first search traversal of the tree is performed while constructing the NFA graph. Each of the sub-expressions creates a sub-graph, which are merged in the main graph using empty transitions. Removing the unnecessary empty transitions further optimizes the final NFA. The optimized NFA for the RE example in Fig. 3 is shown in Fig. 4g.

Fig. 4a — 4f depicts the transformation process for each operator from the RE (or AST) into an equivalent NFA.

Using the *Subset Construction Algorithm* [15], the optimized NFA is converted into an equivalent DFA. During this transformation, the PaREM creates a log file with the transition

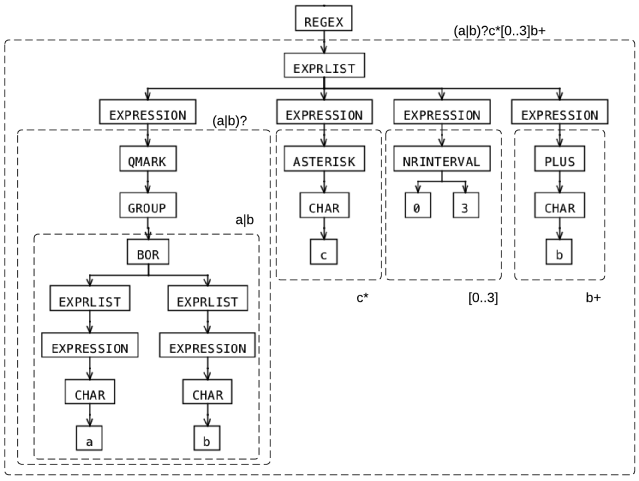
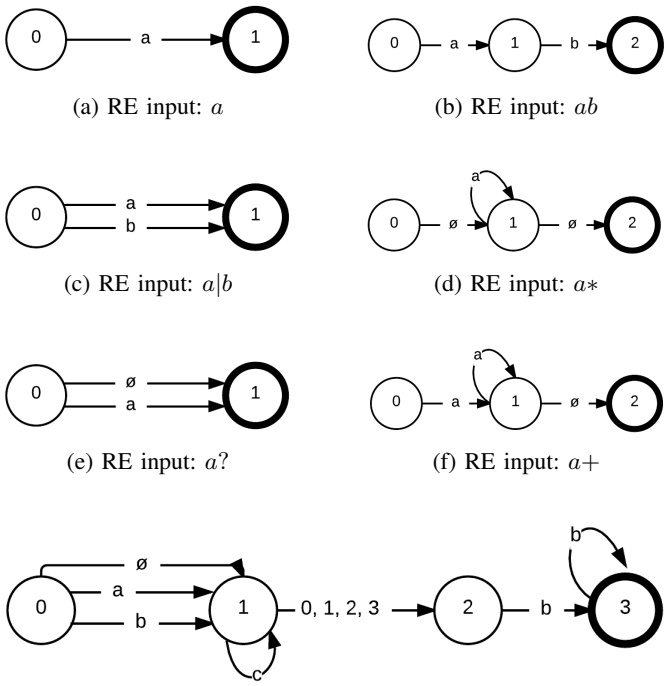


Fig. 3: Abstract Syntax Tree representation for $(a|b)?c * [0..3]b+$ RE



(a) RE input: a (b) RE input: ab

(c) RE input: $a|b$ (d) RE input: $a*$

(e) RE input: $a?$ (f) RE input: $a+$

(g) RE input: $(a|b)?c * [0..3]b+$

Fig. 4: Transformation of RE operators into NFA

table. Theoretically, the DFAs number of states may have an exponential relationship to the NFAs number of states, which leads to the well-known state explosion issue. However, most of the real-world NFA produce a DFA with approximately the same number of states.

Finally, from the DFA we generate executable source code that implements the REM for the corresponding DFA [14] [16]. There are different possible ways of representing a DFA, but we have selected two different forms: (1) *if-else* statements, and (2) *graphs*.

The *if-else* approach is a straightforward way of implementing a DFA. This approach creates an if-statement for each transition of the automaton. However, this approach is not recommended for large automatons. The *if-else* approach provides a sequential solution for regular expression matching.

The *graph-based* approach provides an easy way to add/remove transitions or states in the automaton, and consequently reduces the risk of having incorrect representation of the automaton.

For *graph-based* representation in the source code, we have used an *adjacency matrix*, which represents the transition table. This approach has faster lookups to check for the presence or absence of a specific transition, compared to the *adjacency list* representation of the automaton. The *graph-based* solution provides the implementation of the parallel regular expression-matching algorithm presented in this paper.

TABLE IV: System Configuration

Operating System	CentOS 6.2 (Linux kernel 2.6.32)
Processor	2× Intel® Xeon® Processor E5-2695 v2 (2.40GHz, 30MB Cache, 12 Cores)
RAM	8 × 16GB
OpenMP	3.1

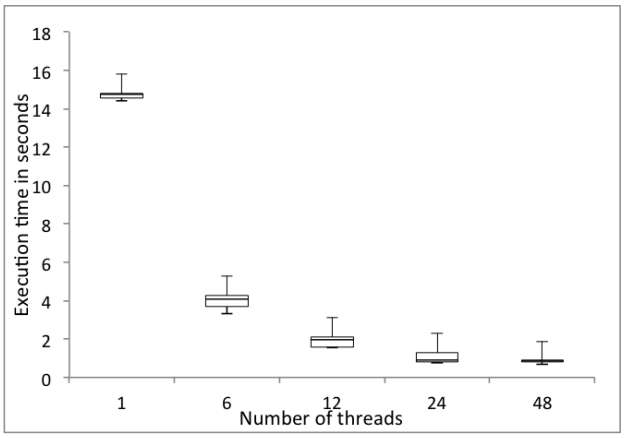
IV. Experimental Evaluation

For experimental purposes, an automaton that finds all occurrences of the word *"parallel"* has been implemented, which results with an automaton with nine states (shown on Fig. 1) and an alphabet of five characters. Table IV lists the major features of experimentation platform. We use a shared-memory system with two 12-core Intel® Xeon® processors of the type E5-2695 v2 for evaluation of our approach. Each of the 12 physical cores supports two threads (also known as logical cores). In total, our system has 24 physical cores or 48 logical cores.

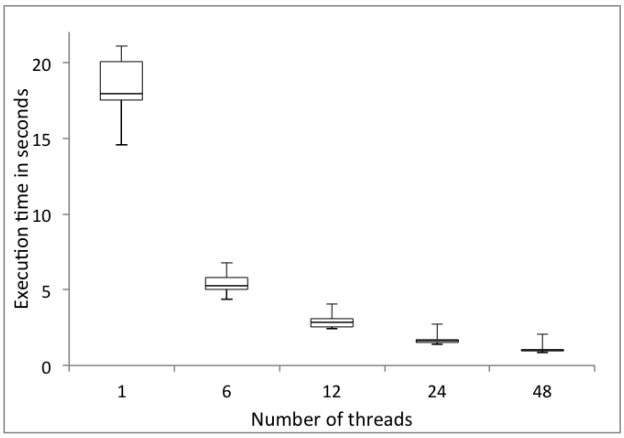
Fig. 5a — 5e depicts the performance results for five problem sizes and various numbers of threads. Each experiment has been repeated 20 times to address the random performance fluctuations. The string length determines the problem size and in our experiment, we used five strings of following lengths: 6.69e+07, 1.34e+08, 2.68e+08, 5.36e+08 and 1.07e+09.

Execution times are shown in Fig. 5a — 5e, whereas the speedup is depicted in Fig. 5f. The speed up for the smallest input length (6.69e+07 characters) in our set of experiments closely follows the linear speedup up to 24 threads (Fig. 5f). For larger input lengths, we may observe noteworthy speedup improvements for 24 and 48 threads. Considering all experiments the highest speedup of $21\times$ was achieved for input length 6.69e+07 characters and 48 threads.

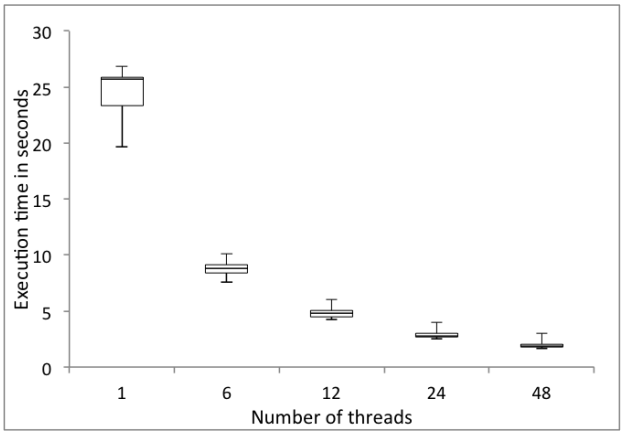
Table V shows the influence of input length in the cache misses and the speedup. We varied the input length using 24 and 48 threads. With the increase of input length, the number



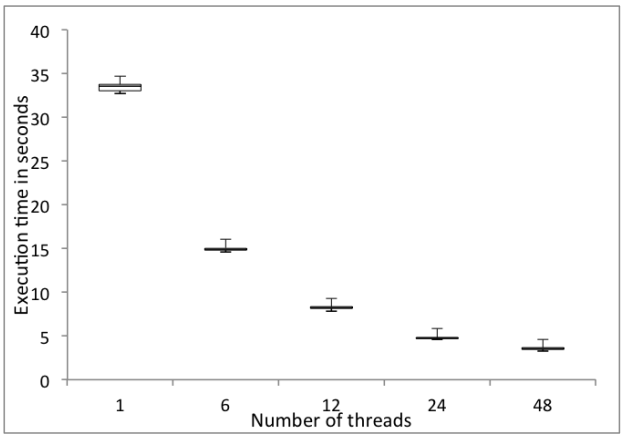
(a) input length: 6.69e+07



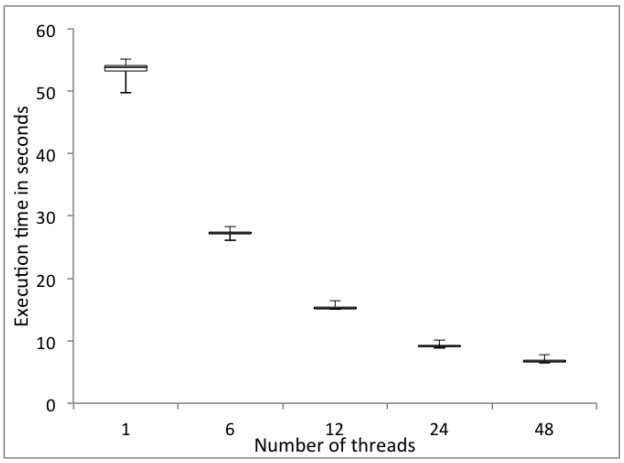
(b) input length: 1.34e+08



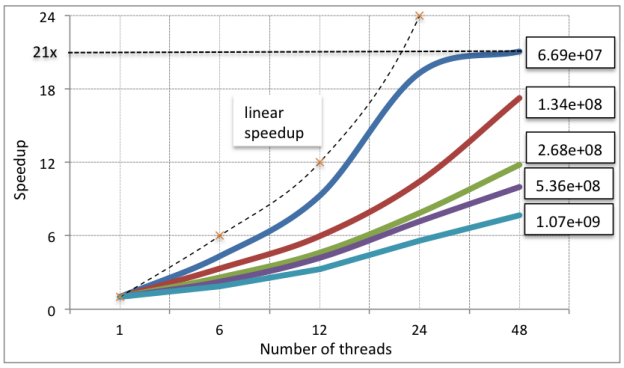
(c) input length: 2.68e+08



(d) input length: 5.36e+08



(e) input length: 1.07e+09



(f) speedup

Fig. 5: Performance results. As input are used five strings of the following lengths: 6.69e+07, 1.34e+08, 2.68e+08, 5.36e+08 and 1.07e+09. Execution times are shown in (a e), whereas the speedup is shown in (f). The speed up for the smallest input length (6.69e+07 characters) in our set of experiments closely follows the linear speedup up to 24 threads. The maximum speedup of $21\times$ is achieved for 48 threads and input string of 6.69e+07 characters.

TABLE V: Influence of Input length in cache misses and speedup for 24 and 48 threads

Input Length	24 threads		48 threads	
	Cache Misses [106]	Speedup	Cache Misses [106]	Speedup
6.69e+07	36.34	19.32	36.76	21.08
1.34e+07	70.15	10.44	71.07	17.27
2.68e+08	167.57	7.87	140.57	11.81
5.36e+08	339.26	7.18	367.07	9.99
1.07e+09	681.71	5.62	716.02	6.69

of cache misses increases and the speedup decreases. For the smallest input length in our set of experiments (6.69e+07 characters) that largely fits in the available cache, using 24 threads, the number of cache misses is 36.34e+06 and the speedup is $19.32\times$. For the largest input length (1.07e+09) we obtained 681.71e+06 cache misses and a speedup of $5.62\times$.

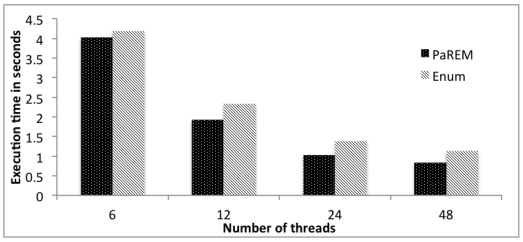
The obtained cache misses for 48 threads are comparable to those for 24 threads (see Table V). For the smallest input length the number of cache misses is 36.76e+06 and the speedup is $21.08\times$. For the largest input length (1.07e+09) we obtained 716.02e+06 cache misses and a speedup of $6.69\times$. We may observe that for all tested input lengths the speedup-gain when 48 logical cores (hyper-threading) are used compared to 24 physical cores.

A. Performance comparison of PaREM algorithm with the General Enumeration Approach

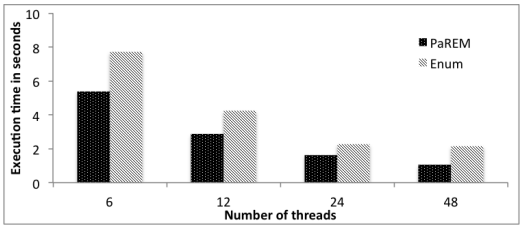
The main difference of the PaREM algorithm and the General Enumeration Approach (Enum) proposed by [6] is the way of speculation of the next set of possible initial states for each chunk of the input string. While the Enum algorithm for general DFAs considers all the states of the automaton as initial states, the PaREM algorithm finds the most accurate initial states. Comparing to PaREM that requires only five calculations to find the correct path, the Enum algorithm requires 28 calculations to be performed in order to find the correct initial states for the example described in section II.C.

We have run the experiment example from section II.C with the same input sizes and number of threads for the General Enumeration approach as well. Figure 6a — 6e depicts the impact of finding the most accurate initial states in the time execution. The sequential version (running in one thread) is the same for both algorithms, because they start the calculations from state q_0 on processing unit P_0. The Enumeration Approach requires more calculations for finite automata with larger number of states, larger input size and for higher number of processing units.

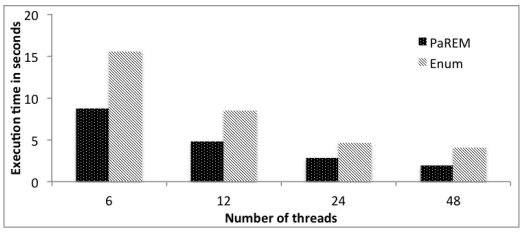
The execution time of the Enumeration Approach compared to the PaREM algorithm increases as we increase either the input size or the number of threads. The execution time of PaREM is $2.3\times$ better than Enum, which is achieved in the largest number of threads (48) and the biggest problem size(1.07e+09), and only $1.04\times$ better than Enum for the smallest number of threads (6) and the smallest input size (6.69+e07).



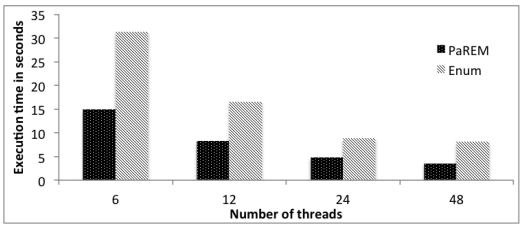
(a) input length: 6.69e+07



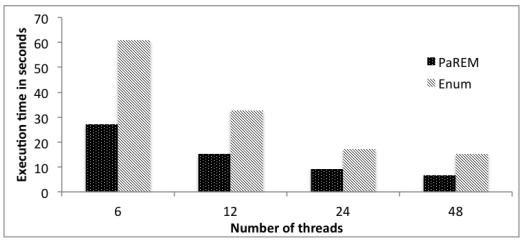
(b) input length: 1.34e+08



(c) input length: 2.68e+08



(d) input length: 5.36e+08



(e) input length: 1.07e+09

Fig. 6: Comparison between PaREM algorithm and General Enumeration Approach.

V. Related Work

Holub and Stekr [6] propose an approach for parallel REM via DFA by splitting the input string in small chunks and running these chunks on each core, but due to pre-calculation of initial states for each sub input, this was not efficient for general DFA. Their algorithm runs efficiently for a specific type of DFA, so called synchronizing automata, that relies on the input automaton being k-local.

Yang and Prassana [7] propose the segmentation of regular expressions and perform the REM evaluation via nondeterministic finite automata. The major aim is to optimize the use of memory hierarchy in case of automata with many states and large transition table. In contrast to our approach, the authors of [7] focus on large automata but do not address specifically algorithmic optimizations with respect to large input strings.

Mytkowicz and Schulte [8] propose an approach that exploits SIMD, instruction and thread level parallelism in the context of finite state machines computations. To increase the opportunities for data-parallelism authors of [11] have devised a method for breaking data-dependencies with enumeration. This approach is not based on speculation with respect to initial state determination.

Kumar et al. [17] address the issue of large-scale finite automata (also known as the state explosion problem) by splitting regular expressions into two parts: (1) a prefix that contains frequently visited parts of the automata, and (2) a suffix that is the rest of the automaton. The aim is to have a small DFA for frequently accessed parts of automata that fits in cache memory.

Luchaup et al. [18] propose an approach of finding the correct initial state by speculation. They believe that guessing the state of the DFA at certain position (network intrusion detection DFA based scanning spends most of the time in a few hot states) has a very good chance that after a few steps will reach the correct state. They validate these guesses using a history of speculated states. In comparison to our algorithm, the convergence of the guessed state and the correct state is not guaranteed. Furthermore, if a thread does not converge on its sub input, then the next thread is forced to start from a new state, which limits the scalability [8].

Our algorithm is based on splitting the input into smaller sub-inputs (domain decomposition); however, we have devised a method to bypass the need of pre-calculation of all initial states by finding the most accurate possible initial states. Our approach is not limited to a particular type of DFA, and is efficient for a large spectrum of input sizes.

In contrast to the related work, our tool is capable of automatically generating a ready to compile and execute code for shared-memory systems, by taking as input a RE or FA.

VI. Summary and Future Work

Regular expression matching is essential for many applications such as lexical analysis, data mining [19], or network security. We have presented a parallel algorithm for regular expression matching that is based on our improved speculative determination of initial states.

Our tool PaREM transforms automatically any regular expression or finite automata into the corresponding parallel code (C++ and OpenMP), and consequently eases the access to the proposed parallel algorithm for the users without background in parallel programming. Preliminary experimental results show that the performance of our algorithm gracefully scales for various string lengths and numbers of threads. For an input string of 6.69e+07 characters, we obtained a speedup of $21\times$ with 48 threads.

In future, we plan to evaluate our approach for other types of problems, such as DNA sequencing or Network Intrusion Detection Systems. We also plan to extend our implementation for heterogeneous systems.

References

[1] A. Nowzari-Dalini, E. Elahi, H. Ahrabian, and M. Ronaghi, "A new dna implementation of finite state machines." *IJCSA*, vol. 3, no. 1, pp. 51–60, 2006.

[2] A. BabuKaruppiah and S. Rajaram, "Deterministic finite automata for pattern matching in fpga for intrusion detection," in *Computer, Communication and Electrical Technology (ICCCET), 2011 International Conference on*, March 2011, pp. 167–170.

[3] R. Kosala, M. Bruynooghe, J. V. den Bussche, and H. Blockeel, "Information extraction from web documents based on local unranked tree automaton inference." in *IJCAI*, G. Gottlob and T. Walsh, Eds. Morgan Kaufmann, 2003, pp. 403–408.

[4] S. Pllana, S. Benkner, E. Mehofer, L. Natvig, and F. Xhafa, "Towards an intelligent environment for programming multi-core computing systems." in *Euro-Par Workshops*, ser. Lecture Notes in Computer Science, vol. 5415. Springer, 2008, pp. 141–151.

[5] T. Mytkowicz and W. Schulte, "Maine: A library for data parallel finite automata," Tech. Rep. MSR-TR-2012-62, July 2012. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=168379

[6] J. Holub and S. Stekr, "On parallel implementations of deterministic finite automata." in *CIAA*, ser. Lecture Notes in Computer Science, S. Maneth, Ed., vol. 5642. Springer, 2009, pp. 54–64.

[7] Y.-H. E. Yang and V. K. Prasanna, "Optimizing regular expression matching with sr-nfa on multi-core systems." in *PACT*, L. Rauchwerger and V. Sarkar, Eds. IEEE Computer Society, 2011, pp. 424–433.

[8] T. Mytkowicz, M. Musuvathi, and W. Schulte, "Data-parallel finite-state machines." Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 2014.

[9] Acceleo, https://www.eclipse.org/acceleo/, accessed: Sep. 2014.

[10] Jflap, http://www.jflap.org, accessed: Sep. 2014.

[11] A. Aho and J. Ullman, *Foundations of Computer Science: C Edition*, ser. Principles of computer science series. W. H. Freeman, 1994.

[12] J. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Cambridge, 1979.

[13] OpenMP Specification, http://www.openmp.org/wp/openmp-specifications, accessed: Sep. 2014.

[14] A. Aho, *Compilers: Principles, Techniques and Tools (for Anna University), 2/e*. Pearson Education India.

[15] C.-H. Chang and R. Paige, "From regular expressions to dfa's using compressed nfa's." *Theor. Comput. Sci.*, vol. 178, no. 1-2, pp. 1–36, 1997.

[16] A. Arora and A. Shefali Bansal, *Comprehensive Computer and Languages*. Laxmi Publications, 2005.

[17] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia." in *ANCS*, R. Yavatkar, D. Grunwald, and K. K. Ramakrishnan, Eds. ACM, 2007, pp. 155–164.

[18] D. Luchaup, R. Smith, C. Estan, and S. Jha, "Speculative parallel pattern matching." *IEEE Transactions on Information Forensics and Security*, vol. 6, no. 2, pp. 438–451, 2011.

[19] R. Trasarti, F. Bonchi, and B. Goethals, "Sequence mining automata: A new technique for mining frequent sequences under regular expressions," in *Data Mining, 2008. ICDM '08. Eighth IEEE International Conference on*, Dec 2008, pp. 1061–1066.