Int J Parallel Prog (2014) 42:456–489
DOI 10.1007/s10766-013-0258-5

A Speculative Parallel DFA Membership Test for Multicore, SIMD and Cloud Computing Environments

Yousun Ko · Minyoung Jung · Yo-Sub Han · Bernd Burgstaller

Received: 16 October 2012 / Accepted: 17 July 2013 / Published online: 28 August 2013

Abstract We present techniques to parallelize membership tests for Deterministic Finite Automata (DFAs). Our method searches arbitrary regular expressions by matching multiple bytes in parallel using speculation. We partition the input string into chunks, match chunks in parallel, and combine the matching results. Our parallel matching algorithm exploits structural DFA properties to minimize the speculative overhead. Unlike previous approaches, our speculation is *failure-free*, i.e., (1) sequential semantics are maintained, and (2) speed-downs are avoided altogether. On architectures with a SIMD gather-operation for indexed memory loads, our matching operation is fully vectorized. The proposed load-balancing scheme uses an off-line profiling step to determine the matching capacity of each participating processor. Based on matching capacities, DFA matches are load-balanced on inhomogeneous parallel architectures such as cloud computing environments. We evaluated our speculative DFA membership test for a representative set of benchmarks from the Perl-compatible Regular Expression (PCRE) library and the PROSITE protein database. Evaluation was conducted on a 4 CPU (40 cores) shared-memory node of the Intel Academic Program Manycore Testing Lab (Intel MTL), on the Intel AVX2 SDE simulator for 8-way fully vectorized SIMD execution, and on a 20-node (288 cores) cluster on the Amazon EC2 computing cloud. Obtained speedups are on the order of $\mathcal{O}\left(1 + \frac{|P|-1}{|Q|\cdot\gamma}\right)$, where $|P|$ denotes the number of processors or SIMD units, $|Q|$ denotes the number of DFA states, and $0 < \gamma \leq 1$ represents a statically computed DFA property. For all observed cases, we found that $0.02 < \gamma < 0.47$. Actual speedups range from 2.3× to 38.8× for up to 512 DFA states for PCRE, and between 1.3× and 19.9× for up to 1,288 DFA states for PROSITE on a 40-core MTL node. Speedups on the EC2 computing cloud range from 5.0× to 65.8× for PCRE, and from 5.0× to 138.5× for PROSITE.

Y. Ko · M. Jung · Y.-S. Han · B. Burgstaller (✉)
Department of Computer Science, Yonsei University, Seoul, Korea
e-mail: bburg@cs.yonsei.ac.kr

Speedups of our C-based DFA matcher over the Perl-based ScanProsite scan tool range from 559.3× to 15079.7× on a 40-core MTL node. We show the scalability of our approach for input-sizes of up to 10 GB.

Keywords DFA membership test · Parallel pattern matching · Parallel regular expression matching · Speculative parallelization · Multicores

1 Introduction

Locating a string within a larger text has applications with text editing, compiler front-ends and web browsers, scripting languages, file-search (grep), command-processors, databases, Internet search engines, computer security, and DNA sequence analysis. Regular expressions allow the specification of a potentially infinite set of strings (or patterns) to search for. A standard technique to perform regular expression matching is to convert a regular expression to a DFA and run the DFA on the input text. DFA-based regular expression matching has robust, linear performance in the size of the input. However, practical DFA implementations are inherently sequential as the matching result of an input character is dependent on the matching result of the previous characters. Related to DFA matching on parallel architectures, considerable research effort has been recently spent [19,23,28,29,40,47].

To speed up DFA matching on parallel architectures, we propose to use speculation. With our method, the input string is divided into chunks. Chunks are processed in parallel using sequential DFA matching. For all but the first chunk, the starting state is unknown.

The core contribution of our method is to exploit structural properties of DFAs to bound the set of initial states the DFA may assume at the beginning of each chunk. Each chunk will be matched for its reduced set of possible initial states. By introducing such a limited amount of redundant matching computation for all but the first chunk, our DFA matching algorithm avoids speed-downs altogether (i.e., the speculation is failure-free [30]). To achieve load-balancing, the input string is partitioned non-uniformly according to processor capacity and work to be performed for each chunk. These properties open up the opportunity for an entire new class of parallel DFA matching algorithms.

We present the time complexity of our matching algorithms, and we conduct an extensive experimental evaluation on SIMD, shared-memory multicore and cloud computing environments. For experiments, we employ regular expressions from the PCRE Library [34] and from the PROSITE protein pattern database [35]. We show the scalability of our approach for input-sizes of up to 10 GB.

The paper is organized as follows. In Sect. 2, we introduce background material. In Sect. 3, we discuss a motivating example for our speculative DFA matching algorithms. In Sect. 4, we introduce our algorithms and their complexity with respect to speedup and costs. Section 5 shows three implementations for SIMD, shared-memory multicore and cloud-computing environments. Section 6 contains experimental results. We discuss the related work in Sect. 7 and draw our conclusions in Sect. 8.

Algorithm 1: Sequential DFA matching

```
Input  : transition function $\delta$, input string $Str$, start state $q_0$, set of final states $F$
Output: true if input is matched, false otherwise
state ← $q_0$
for $i \leftarrow 0$ to $|Str| - 1$ do
    state ← $\delta$(state, $Str[i]$)
if state $\in F$ then
    return true;                                   // input matched
return false
```

2 Background

2.1 Finite Automata

Let Σ denote a finite alphabet of characters and Σ^* denote the set of all strings over Σ. Cardinality $|\Sigma|$ denotes the number of characters in Σ. A language over Σ is any subset of Σ^*. The symbol $\emptyset$ denotes the empty language and the symbol λ denotes the null string. A finite automaton A is specified by a tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is an input alphabet, $\delta : Q \times \Sigma \rightarrow 2^Q$ is a transition function, $q_0 \in Q$ is the start state and $F \subseteq Q$ is a set of final states. We define A to be a DFA if δ is a transition function of $Q \times \Sigma \rightarrow Q$ and $\delta(q, a)$ is a singleton set for any $q \in Q$ and $a \in \Sigma$. Let $|Q|$ be the number of states in Q. We extend transition function δ to δ^*: $\delta^*(q, ua) = p \Leftrightarrow \delta^*(q, u) = q'$, $\delta(q', a) = p$, $a \in \Sigma$, $u \in \Sigma^*$. We assume that a DFA has a unique error (or sink) state q_e.

An input string Str over Σ is accepted by DFA A if the DFA contains a labeled path from q_0 to a final state such that this path reads Str. We call this path an accepting path. Then, the language $L(A)$ of A is the set of all strings spelled out by accepting paths in A.

The DFA membership test determines whether a string is contained in the language of a DFA. The DFA membership test is conducted by computing $\delta^*(q_0, Str)$ and checking whether the result is a final state. Algorithm 1 denotes the sequential DFA matching algorithm. As a notational convention, we denote the symbol in the ith position of the input string by $Str[i]$.

2.2 Amazon EC2 Infrastructure

The Amazon Elastic Computing Cloud (EC2) allows users to rent virtual computing nodes on which to run applications. EC2 is very popular among researchers and companies in need of instant and scalable computing power. Amazon EC2 provides resizable compute capacity where users pay on an hourly basis for launched (i.e., up-and-running) virtual nodes. By using virtualized resources, a computing cloud can serve a much broader user base with the same set of physical resources. Amazon EC2 virtual computing nodes are virtual machines running on top of a variant of the Xen hypervisor. To create a virtual machine, EC2 provides machine images which contain a pre-configured operating system plus application software. Users can adapt

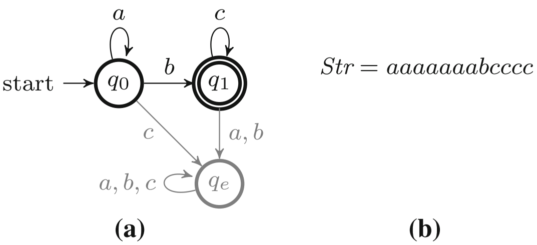


Fig. 1 Example DFA including the error state q_e(**a**) and 12-symbol input string (**b**)

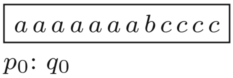


Fig. 2 Notation: example input matched by processor p_0 starting in DFA state q_0

machine images prior to deployment. The launch of a machine image creates a so-called instance, which is a copy of the machine image executing as a virtual server in the cloud. To provide a unit of measure for the compute capacities of instances, Amazon introduced so-called EC2 Compute Units (CUs), which are claimed to provide the equivalent CPU capacity of a 1.0–1.2 GHz 2007 Opteron or 2007 Xeon processor [3]. Because there exist many such CPU models in the market, the exact processor capacity equivalent to one CU is not entirely clear. Instance types are grouped into nine families, which differ in their processor, I/O, memory and network capacities. Instances are described in [3]; the instances employed in this paper are outlined in Sect. 5. To create a cluster of EC2 instances, the user requires the launch of one or more instances, for which the instance type and the machine image must be specified. The user can specify any machine image that has been registered with Amazon, including Amazon's or the user's own images. Once instances are booted, they are accessible as computing nodes via ssh. A maximum of 20 instances can be used concurrently, but this limit may be increased upon user request [4].

3 Overview

The core idea behind our speculative DFA matching method is to divide the input into several chunks and process them in parallel. As a motivating example we consider the DFA depicted in Fig. 1. This DFA accepts strings which contain zero or more occurrences of the symbol a, followed by exactly one occurrence of symbol b, followed by zero or more occurrences of symbol c. For the exposition of this motivating example we have included the DFA's error state q_e and its adjacent transitions, which are depicted in gray. The DFA's alphabet is $\Sigma = \{a, b, c\}$, and we consider the 12-symbol input string from Fig. 1b.

Assuming that it takes on the order of one time-unit to process one character from the input string, Algorithm 1 will spend 12 time units for the sequential membership test. This is denoted by the notation in Fig. 2, where a processor p_0 matches the input string from Fig. 1b. The DFA is in state q_0 initially.

To parallelize the membership test for three processors, the input string from Fig. 2 can be partitioned into three chunks of four symbols each, and assigned to processors p_0, p_1 and p_2 as illustrated in Fig. 3.

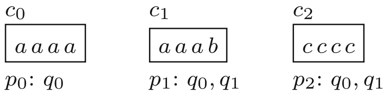


Fig. 3 Dividing the input into equal-sized chunks

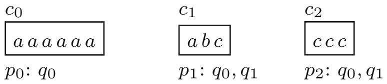


Fig. 4 Balanced input partition according to the number of states matched by a processor

Because the DFA will initially be in start state q_0, the first chunk (c_0) needs to be matched for q_0 only. For all subsequent chunks, the DFA state at the beginning of the chunk is initially unknown. Hence, we use speculative computations to match subsequent chunks for all states the DFA may assume. We will discuss in Sect. 4 how the amount of speculative computations can be kept to a minimum. For our motivating example, we assume the DFA to be in either state q_0 or q_1 at the beginning of chunks c_1 and c_2. As depicted by the partition from Fig. 3, processor p_0 will match chunk c_0 for state q_0, whereas processors p_1 and p_2 will match their assigned chunks for both q_0 and q_1. To match a chunk for a given state, a variation of the matching loop (lines 1–3) of Algorithm 1 is employed.

After processors p_0, p_1 and p_2 have processed their assigned chunks in parallel, the results from the individual chunks need to be combined to derive the overall result of the matching computation. Combining proceeds from the first to the last chunk by propagating the resulting DFA state from the previous chunk as the initial state for the following chunk. According to Fig. 1, the DFA from our motivating example will be in state q_0 after matching chunk c_0. State q_0 is propagated as the initial state for chunk c_1. Processor p_1 has matched chunk c_1 for both possible initial states, i.e., q_0 and q_1, from which we obtain that state q_0 at the beginning of chunk c_1 takes the DFA to state q_1 at the end of chunk c_1. Likewise, the matching result for chunk c_2 is now applied to derive state q_1 as the final DFA state.

To compute the speedup over sequential DFA matching, we note that processor p_0 processes 4 input characters, whereas processors p_1 and p_2 match the assigned chunks twice, for a total of 8 characters per processor. The resulting speedup is thus $\frac{12}{8}$ or 1.5 (Combining the matching results will induce slight additional costs on the order of the number of chunks, as we will consider in Sect. 4).

An input partition that accounts for the work imbalance between the initial and all subsequent chunks is depicted in Fig. 4. Because processors p_1 and p_2 match chunks for two states each, their chunks are only half the size of the chunk assigned to processor p_0. All processors now process 6 characters each, resulting in a balanced load and a 2x speedup over sequential matching.

By considering the structure of DFAs, the amount of redundant, speculative computation can be reduced. For the DFA in Fig. 1, we observe that for each alphabet character $x \in \Sigma = \{a, b, c\}$, there is only one DFA state (except the error state q_e) with an incoming transition labeled x. Thus, this particular DFA has the structural property that for any given character $x \in \Sigma$, the DFA state after matching character x is known *a-priory* to be either the error state or the state with the incoming transition labeled x.

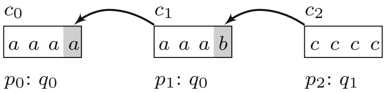


Fig. 5 Balanced input partition considering one symbol reverse lookahead

A processor can exploit this structural DFA property by performing a *reverse lookahead* to determine the last character from the previous chunk. From this character the DFA state at the beginning of the current chunk can be derived. In Fig. 5, the reverse lookahead for our motivating example is shown. Reverse lookahead characters are shaded in gray. Character a is the lookahead character in chunk c_0; only DFA state q_0 from Fig. 1 has an incoming transition labeled a, thus the DFA must be in state q_0 at the beginning of chunk c_1. Likewise, the DFA must be in state q_1 at the beginning of chunk c_2, because state q_1 is the only DFA state with an incoming transition labeled b (the lookahead character of chunk c_1). Note that for these considerations the error state q_e can be ignored, because once a DFA has reached the error state, it will stay there (e.g., see Fig. 1). Thus, to compute the DFA matching result it is unnecessary to process the remaining input characters once the error state has been reached.

Because now all processors have to match only a single state per chunk, the chunks are of equal size. For three processors, we achieve a speedup of 3× over sequential matching for the motivating example.

It should be noted that in the general case the structure of DFAs will be less ideal, i.e., there will be more than one state with incoming transitions labeled by a particular input character. Consequently, each chunk will have to be matched for more than one DFA state. We will develop a measure for the suitability of a DFA for this type of speculative parallelization in Sect. 4. Our analysis of the time-complexity of this method shows that for $|P| > 1$, a speedup is achievable in general. This has been confirmed by our experimental evaluations on SIMD, shared-memory multicore, and the Amazon EC2 cloud-computing environments. We will discuss the trade-offs that come with multi-character reverse lookahead, and we will incorporate inhomogeneous compute capacities of processors to resolve load imbalances. This is essential to effectively utilize heterogeneous multicore architectures, and to overcome the performance variability of nodes reported with cloud computing environments [5,41].

4 Speculative DFA Matching

Our speculative DFA matching approach is a general method, which allows a variety of algorithms that differ with respect to the underlying hardware platform and the incorporation of structural DFA properties. We start this section with the formalization of our basic speculative DFA matching example from Sect. 3. We then present our approach to exploit structural DFA properties to speed up parallel, speculative DFA matching. Section 5 contains variants tailored for SIMD, shared memory multicores and cloud computing environments.

4.1 The Basic Speculative DFA Matching Algorithm

Our parallel DFA membership test consists of the following four steps; the first step is only required on platforms with processors of inhomogeneous performance.

1. Offline profiling to determine the DFA matching capacity of each participating processor,
2. partitioning the input string into chunks such that the utilization of the parallel architecture is maximized,
3. performing the matching process on chunks in parallel such that redundant computations are minimized, and
4. merging partial results across chunks to derive the overall result of the matching computation.

4.1.1 Offline Profiling

For environments with inhomogeneous compute capacities, our offline profiling step determines the DFA matching capacities of all participating processors. This information is required to partition work equally among processors and thus balance the load. With heterogeneous multicore hardware architectures such as the Cell BE [16], offline profiling must be conducted only once to determine the performance of all types of processor cores provided by the architecture. With cloud computing environments such as the Amazon EC2 cloud [3], users only have limited control on the allocation of cloud computing nodes. Moreover, the performance of cloud computing nodes has been found to differ significantly, which is by a large extent attributed to variations in the employed hardware platforms [5,41]. To compensate for the performance variations between cloud computing nodes, offline profiling will be conducted at cluster startup time. Profiling cluster nodes in parallel takes only on the order of milliseconds, which makes the overhead from profiling negligible compared to the substantial cluster startup times on EC2 (on the order of minutes by our own experience and also reported in [33]).

To account for performance variations, we introduce a weight factor w_k, which denotes the processor capacity of a processor p_k, normalized by the average processor capacity of the system. On each processor p_k, our profiler performs several partial sequential DFA matching runs for a predetermined number of input symbols on a given benchmark DFA. From the median of the obtained execution times, we compute the number of symbols m_k matched by processor p_k per microsecond. The processor's weight factor w_k is then computed as

$$w_k = m_k \cdot \left(\frac{1}{|P|} \cdot \sum_{0 \leq i < |P|} m_i \right)^{-1}. \tag{1}$$

Columns "m_k" and "w_k" of Table 1 contain example matching capacities and corresponding weights for a system of three processors. We will apply processor weights to partition the input string into chunks as follows.

4.1.2 Input Partitioning

We observed already with our motivating example from Fig. 3 that partitioning the input into equal-sized chunks will result in load-imbalance: because for the first chunk the initial DFA state is known to be q_0, the first chunk needs to be matched only once. All other chunks must be matched for all possible initial states of the chunk, i.e., $|Q|$ times, in the worst case. In what follows, we will derive a partition of the input *Str* into $|P|$ chunks, assuming that all except the first chunk need to be matched for $|Q|$ states. In Sect. 4.2, we will exploit structural DFA properties to reduce the number of states to be matched per chunk.

Intuitively, because processor p_0 has to match chunk c_0 only once, it can process a larger portion of the input *Str* than the processors assigned to subsequent chunks. (This was observed already in Fig. 4, where chunk sizes were adjusted such that all processors processed the same number of characters from the input). The objective of our optimization is to determine chunk sizes in such a way that the processing times for all chunks are equal. The purpose of the following equations is to compute a partition of the input into chunks c_i, $0 \leq i < |P|$, where chunk c_i is a sequence of symbols from the input allocated to processor p_i.

Let L_i denote the length of chunk c_i when $0 \leq i < |P|$, and n be the length of the input *Str*. Let us further assume that matching of a character from the input takes constant time. Processor p_0 matches chunk c_0 from starting state q_0. All other chunks need to be matched for all possible initial states. To keep work among processors balanced, chunk c_0 must be $|Q|$ times longer than the other chunks, i.e., it must hold that

$$L_i = \frac{L_0}{|Q|}, \text{ for } 1 \leq i < |P|. \tag{2}$$

The lengths of all chunks must add up to n, namely

$$\sum_{0 \leq i < |P|} L_i = n. \tag{3}$$

If processors have non-uniform processing capacity, we incorporate weight factors from Eq. (1), such that weighted chunk sizes must add up to n.

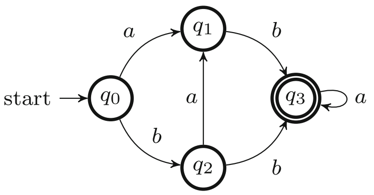
$$\sum_{0 \leq i < |P|} L_i w_i = n. \tag{4}$$

Finally we solve the unknown L_0 by substituting corresponding parts of Eq. (2) and (1) in Eq. (4), i.e.,

$$L_0 = \frac{n \cdot |Q|}{w_0 \cdot |Q| + \sum_{1 \leq i < |P|} w_i}. \tag{5}$$

The start and end positions for each chunk c_k, $0 \leq k < |P|$, are computed by the following equations. (Note that for $k = 1$, the range for the sum over $L_0 w_i$ is 0.)

Fig. 6 Example DFA (**a**) and input string with 36 symbols (**b**)



(a)

$Str = bababbababbababbaaabbababbbaabbaaaba$

(b)

Table 1 Computation of chunk sizes for Fig. 6 and three processors of non-uniform processing capacities

Processor	m_k	w_k	$L_0 \cdot w_k$	Input character range
p_0	50	1.5	28.8	0–27
p_1	25	0.75	3.6	28–31
p_2	25	0.75	3.6	32–35

$$\text{StartPos}(c_k) = \begin{cases} 0, & \text{for } k = 0, \\ \left\lfloor L_0 w_0 + \frac{1}{|Q|} \sum_{1 \le i < k} L_0 w_i \right\rfloor & \text{otherwise} \end{cases} \tag{6}$$

$$\text{EndPos}(c_k) = \begin{cases} n - 1, & \text{for } k = |P| - 1, \\ \lfloor L_0 w_0 + \frac{1}{|Q|} \sum_{1 \le i \le k} L_0 w_i \rfloor - 1, & \text{otherwise} \end{cases} \tag{7}$$

An example DFA and an input string of length $n = 36$ are presented in Fig. 6. The corresponding chunk sizes for three processors with different processing capacities are depicted in Table 1. We observe by Eq. (5) that the length L_0 of chunk c_0 is 19.2 characters, and the weighted length according to processor weight w_0 is 28.8 characters. From Eq. (2) we observe that the remaining chunks are four times shorter than chunk c_0, because they have to be matched for $|Q| = 4$ states. The weighted lengths of chunks c_1 and c_2 are thus 3.6 characters each. The rightmost column of Table 1 depicts the character ranges of the input as they have been assigned to each chunk.

4.1.3 Matching of Chunks

Algorithm 2 depicts our basic speculative DFA matching procedure. We employ the notation introduced in [19] to denote a mapping of *possible initial states* to *possible last active states* of a chunk. This mapping is required to store a chunk's matching results for all possible initial states. After matching chunks in parallel, the computed mappings will be used to derive the overall DFA matching result. Formally, this mapping is defined as a vector

$$\mathcal{L}_i = [l_0, l_1, \ldots, l_{|Q|-1}],$$

where $0 \le i < |P|$ and $l_j \in Q$ for all $0 \le j < |Q|$. Let element l_j of $\mathcal{L}_i$ denote the last active state, assuming that processor p_i starts in state q_j and processes the DFA membership test on chunk c_i, i.e., $\delta^*(q_j, c_i) = l_j$.

As an example, we consider chunk c_2 from Fig. 3 and the DFA from Fig. 1. Chunk c_2 will be matched for the possible initial states q_0 and q_1, with the resulting last active states q_e and q_1 and the result vector $\mathcal{L}_2 = [q_e, q_1]$. The meaning of vector $\mathcal{L}_2$ is that if the DFA assumes state q_0 at the beginning of chunk c_2, then it will be in state q_e after matching chunk c_2. If the DFA assumes state q_1 at the beginning of chunk c_2, then it will be in state q_1 after matching chunk c_2.

Our basic speculative DFA matching procedure employs Eqs. (6) and (7) to derive the start and end position of each chunk (lines 4–5 of Algorithm 2). The algorithm distinguishes between the first input chunk (lines 6–8) and all subsequent chunks (lines 9–12). According to our partitioning scheme, chunk c_0 is only matched for the start state q_0 (lines 7–8). For all subsequent chunks c_i, all possible DFA states are matched and stored in vector $\mathcal{L}_i$. Chunk sizes are chosen according to processor weights and the number of states to be matched with each chunk. The goal of this partitioning is to load-balance the DFA matching to effectively utilize the underlying parallel hardware platform. We will discuss in Sect. 4.4 that our partitioning scheme makes this speculation failure–free. The output of Algorithm 2 is the set of vectors $\mathcal{L}_i$, where each vector describes the possible last states according to the possible initial states of a given chunk.

Algorithm 2: Basic speculative DFA matching

```
   Input  : $\delta$, $Q$, $\Sigma$, $P$, $Str = c_0c_1 \ldots c_{|P|-1}$
   Output: vector $\mathcal{L}_i$ for each chunk $c_i$
1  for $i \leftarrow 0$ to $|P| - 1$ do in parallel
2      for $j \leftarrow 0$ to $|Q| - 1$ do
3          $\mathcal{L}_i[j] \leftarrow j$ ;                              // initialize vector $\mathcal{L}_i$
4      Start $\leftarrow$ StartPos($c_i$)
5      End $\leftarrow$ EndPos($c_i$)
6      if $i = 0$ then                                          // chunk $c_0$
7          for $k \leftarrow$ Start to End do
8              $\mathcal{L}_0[0] \leftarrow \delta(\mathcal{L}_0[0], Str[k])$
9      else                                                     // chunks $c_1 \ldots c_{|P|-1}$
10         foreach $j \in Q$ do
11             for $k \leftarrow$ Start to End do
12                 $\mathcal{L}_i[j] \leftarrow \delta(\mathcal{L}_i[j], Str[k])$
```

4.1.4 Merging of Partial Results

After matching chunks in parallel, each processor p_i has constructed a mapping $\mathcal{L}_i$ of possible initial states to last active states. To finish the DFA run, the partial results computed for chunks c_i need to be combined to determine the last active state for the DFA-run over the whole input string $Str = c_0c_1 \ldots c_{|P|-1}$. Chunk c_0 is the only chunk for which we know the initial state of the automaton, i.e., q_0. We use this information to apply the mappings $\mathcal{L}_i$ sequentially to derive the last active state as follows (it should be noted that index 0 of the $\mathcal{L}[\ldots]$ mapping is the index of the start state q_0):

$$\text{last active state} = \mathcal{L}_{|P|-1}[\mathcal{L}_{|P|-2}[\ldots \mathcal{L}_0[0] \ldots]]. \tag{8}$$

It has been shown in [19] how a binary reduction (see [27]) can be used to parallelize this computation. A binary reduction uses a combining operation on two maps $\mathcal{L}_i$ and $\mathcal{L}_j$ to derive the combined map $\mathcal{L}_{i,j}$ as depicted in Eq. (9).

$$\mathcal{L}_{i,j} = \begin{bmatrix} \mathcal{L}_j[\mathcal{L}_i[0]] \\ \mathcal{L}_j[\mathcal{L}_i[1]] \\ \vdots \\ \mathcal{L}_j[\mathcal{L}_i[|Q|-1]] \end{bmatrix} \tag{9}$$

The reduction step above can be performed repeatedly in parallel to combine maps until we finally arrive at the map $\mathcal{L}_{0,|P|-1}$ which represents the overall effect of a DFA. In particular, the value $\mathcal{L}_{0,|P|-1}[0]$ will be the last active state of a DFA's run on the input *Str*.

The work in [19] does not provide an evaluation of the relative merits of sequential versus parallel merging of $\mathcal{L}$-vectors. In particular, the details of the employed parallel reduction algorithm are not specified. We conducted experiments on a 40-core shared memory node of the Intel MTL using a binary tree for the parallel reduction to find that the computation associated with the merging of $\mathcal{L}$-vectors is not large enough to justify the overhead of a parallel reduction. Especially the overhead from the synchronization required between each of the $\mathcal{O}(\log_2(|P|))$ reduction steps is costly.

Moreover, the overhead becomes significant if communication cost between nodes are introduced such as with cloud computers. We describe our findings on the overheads of intra-node and inter-node communication with the EC2 computing cloud in detail in Sect. 5. Section 5 introduces a new $\mathcal{L}$-vector merging technique to cope with the overhead on cloud computers.

In short, we applied the sequential merging from Eq. (8) with shared-memory multicore architectures and a new hierarchical merging technique for cloud computing architectures, which will be explained in Sect. 5.

4.2 Optimizations Based on Structural DFA Properties

The amount of work associated with a given chunk is determined by (1) the length of the chunk, and (2) the number of DFA states for which the chunk needs to be matched. In the following, we will distinguish between the initial chunk c_0, and *subsequent* chunks c_i, $i > 0$. Before matching the initial chunk c_0, the DFA will be in the starting state q_0, thus chunk c_0 only needs to be matched for q_0. Prior to the matching of subsequent chunks, the DFA may assume any state in the general case, thus subsequent chunks need to be matched $|Q|$ times (see, e.g., the motivating example in Fig. 3). In this section we will exploit structural properties of DFAs to deduce a potentially smaller number $\mathcal{I}_{\max} \leq |Q|$ of states which is the upper bound of initial states for all subsequent chunks.

The best case, i.e., $\mathcal{I}_{\max} = 1$, has already been observed with our motivating example DFA from Fig. 1. For each character $\sigma \in \Sigma$ of this DFA, it holds that there is only one state targeted by a transition labeled σ. Irrespective of the particular input character σ, the DFA can only assume a single state after matching character σ.

(As mentioned previously, for these considerations we may safely disregard the error state q_e, because from the error state no other state is reachable; thus, a DFA that reached the error state will stay there.) If there is only one possible DFA state after matching an input character, it follows that the DFA can only be in one state after matching the last character prior to each subsequent chunk. Thus the DFA can only be in one possible state at the beginning of each subsequent chunk, and we have $\mathcal{I}_{\max} = 1$.

In the general case, values for $\mathcal{I}_{\max}$ can range between 1 and $|Q|$. In the remainder of this section, we will investigate how to deduce this $\mathcal{I}_{\max}$ value for a particular DFA, and how this information can be incorporated with our speculative DFA matching algorithm. We will consider real-world DFAs from PCRE and PROSITE to find that for all considered DFAs it holds that $\mathcal{I}_{\max} < |Q|$, and that this property can be used to improve DFA matching performance. We have already observed with the input partition in Fig. 5 that reducing the number of initial states of subsequent chunks enables us to increase the sizes of subsequent chunks. Larger subsequent chunks will reduce the size of the initial chunk c_0 in turn. Because we adjust chunk sizes such that all chunks will be processed in the same amount of time, reducing the size of the initial chunk c_0 will reduce the overall execution time of the matching process. The overarching reason for this performance improvement is that the reduction of potential initial states reduces the total number of symbols that have to be matched per chunk.

This can be formalized as follows. Let $\mathcal{I}_{\max}$ denote the maximum number of possible initial states that the DFA may assume at the start over all subsequent chunks. This maximum can be different for each chunk, depending on the last character of the preceding chunk. We assume that for $\mathcal{I}_{\max}$ we pick the maximum value out of all possible sets of initial states over all chunks. If $\mathcal{I}_{\max} < |Q|$, then the length L_0 of chunk c_0 reduces by Eq. (5):

$$\begin{aligned} L_0 &= \frac{n \cdot \mathcal{I}_{\max}}{w_0 \cdot \mathcal{I}_{\max} + \sum_{1 \le i < |P|} w_i} \\ &< \frac{n \cdot |Q|}{w_0 \cdot |Q| + \sum_{1 \le i < |P|} w_i}. \end{aligned} \tag{10}$$

To deduce the maximum value for $\mathcal{I}_{\max}$, we eliminate states that can never be the initial state for a given chunk. For each character $\sigma \in \Sigma$, a DFA will contain a number of states that have an incoming transition labeled σ. Thus, if the last character of a chunk's preceding chunk is σ, then only the states with an incoming transition labeled σ need to be matched. We call the last input character of a chunk's preceding chunk the *reverse lookahead symbol*. The number of states to be matched for a reverse lookahead symbol $\sigma \in \Sigma$ is a static property of a DFA. It will range between 1 and $|Q|$. The maximum number of states to be matched over any reverse lookahead symbol constitutes an upper bound on $\mathcal{I}_{\max}$, i.e., an upper bound on the number of states to be matched for any subsequent chunk. Because $\mathcal{I}_{\max}$ is a static DFA property, we can use it to partition the input into chunks according to Eq. (10). At run-time, a processor will use the reverse lookahead symbol to determine the initial states to be matched for its assigned chunk.

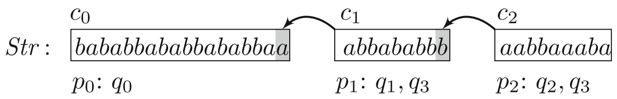


Fig. 7 Partitioned input string with reverse lookahead symbols and set of initial states to be matched for each chunk

Given lookahead symbol σ, we define the set of initial states $\mathcal{I}_\sigma$ as the set of all states that have an incoming transition labeled σ.

$$\mathcal{I}_\sigma = \{s : \delta(x, \sigma) = s\}, \quad \forall s, x \in Q. \tag{11}$$

If symbol σ is the reverse lookahead symbol of chunk c_i, then the set of possible initial states for chunk c_i is $\mathcal{I}_\sigma$. We compute the set of possible initial states for all symbols from the DFA's alphabet Σ and set $\mathcal{I}_{\max}$ to the maximum cardinality among those sets, i.e.,

$$\mathcal{I}_{\max} = \max_{\sigma \in \Sigma} (|\mathcal{I}_\sigma|) . \tag{12}$$

As an example, consider the DFA from Fig. 6a. Figure 7 shows the input string partitioned for three processors of equal capacity, i.e., $w_0 = w_1 = w_2 = 1$. The reverse lookahead symbols are depicted in gray. No reverse lookahead is required for chunk c_0, which will be matched from the DFA's start state q_0. Because the reverse lookahead symbol σ of chunk c_1 is an 'a', upon matching of chunk c_1 the DFA can only be in a state that has an incoming transition labeled 'a'. Likewise, because the reverse lookahead symbol of chunk c_2 is 'b', the DFA can only be in a state that has an incoming transition labeled 'b' upon matching of chunk c_2. We get $\mathcal{I}_a = \{q_1, q_3\}$, $\mathcal{I}_b = \{q_2, q_3\}$, and $\mathcal{I}_{\max} = 2$. Inserting $n = 36$, $\mathcal{I}_{\max} = 2$, $|Q| = 4$ and $w_0 = w_1 = w_2 = 1$ in Eq. (10) yields $L_0 = 18 < 24$ and a speedup of $\frac{24}{18} = 1.\dot{3}$ over the non-optimized matching procedure.

Algorithm 3 applies initial state sets with the DFA matching procedure. Lines 1–7 compute initial state sets $\mathcal{I}_\sigma$ from Eq. (11) and $\mathcal{I}_{\max}$ from Eq. (12). Unlike Algorithm 2, the partitioning is now based on the maximum number of possible initial states, $\mathcal{I}_{\max}$, instead of $|Q|$. The StartPos and EndPos functions that compute the start and end position of each chunk now receive $\mathcal{I}_{\max}$ as the second argument (lines 11–12 in Algorithm 3). We updated Eqs. (6) and (7) to include an additional parameter to pass $\mathcal{I}_{\max}$. In Eqs. (5)–(7), instead of $|Q|$ we then use the provided argument value to partition the input string and to compute the start and end position of each chunk.

Because the maximum number of initial states $\mathcal{I}_{\max}$ is a static property of a DFA, it can be computed off-line. The overhead to compute $\mathcal{I}_{\max}$ can thus be avoided with DFAs that are matched multiple times. For example, with protein patterns maintained in databases, corresponding DFAs can be expected to be matched on several DNA sequences. However, with all our experiments, we computed $\mathcal{I}_{\max}$ online for every matching run (as stated in Algorithm 3), to account for the general case were a DFA is matched only once.

Algorithm 3: DFA matching applying initial state sets

```
Input  : $\delta$, $Q$, $\Sigma$, $P$, $Str = c_0 c_1 \ldots c_{|P|-1}$, $q_0$
Output: vector $\mathcal{L}_{p_i}$ for each chunk $c_i$
foreach $\sigma_i \in \Sigma$ do
    $\mathcal{I}_{\sigma_i} \leftarrow \emptyset$
    foreach $s \in Q$ do
        $q_{\text{target}} \leftarrow \delta(s, \sigma_i)$
        if $q_{target} \neq q_e$ then
            $\mathcal{I}_{\sigma_i} \leftarrow \mathcal{I}_{\sigma_i} \cup q_{\text{target}}$

$\mathcal{I}_{\max} \leftarrow \max(\mathcal{I}_{\sigma_0}, \ldots, \mathcal{I}_{\sigma_{|\Sigma|-1}})$
for $i \leftarrow 0$ to $|P| - 1$ do in parallel
    for $j \leftarrow 0$ to $|Q| - 1$ do
        $\mathcal{L}_i[j] \leftarrow j$                          // initialize vector $\mathcal{L}_i$
    Start $\leftarrow$ StartPos($c_i, \mathcal{I}_{\max}$)
    End $\leftarrow$ EndPos($c_i, \mathcal{I}_{\max}$)
    if $i = 0$ then                                       // chunk $c_0$
        for $k \leftarrow$ Start to End do
            $\mathcal{L}_0[0] \leftarrow \delta(\mathcal{L}_0[0], Str[k])$
    else                                              // chunks $c_1 \ldots c_{|P|-1}$
        foreach $j \in \mathcal{I}_{c_i}$ do
            for $k \leftarrow$ Start to End do
                $\mathcal{L}_i[j] \leftarrow \delta(\mathcal{L}_i[j], Str[k])$
```

Another possible optimization of Algorithm 3 concerns the distribution of cardinalities of initial state sets $\mathcal{I}_\sigma$. If the maximum value $\mathcal{I}_{\max}$ is significantly larger than the average, then it is desirable to divide the input at boundaries with reverse lookahead symbols that have a small initial state set. This would further decrease the number of possible initial states of subsequent chunks. However, searching the input for the occurrence of particular characters constitutes an effort similar to the matching process itself. Moreover, relying on statistical properties of the input string (i.e., the occurrence of particular characters in the input) may violate the failure-freedom of our speculation: if a reverse lookahead symbol with a low set of initial states cannot be found, then additional states need to be matched, resulting in a possible speed-down. In contrast, by considering $\mathcal{I}_{\max}$ states, our optimization always shows equal or better performance than the non-optimized matching procedure that has to match all states in Q.

4.3 Multiple Reverse Lookahead Symbols

As discussed in the previous section, a smaller $\mathcal{I}_{\max}$ constant will decrease the number of symbols to be matched per chunk, thereby increasing DFA matching performance. We can potentially decrease the number of possible initial states, if we employ additional reverse lookahead symbols with each chunk. Given a string of reverse lookahead symbols $\sigma_1 \ldots \sigma_k$, $k \geq 1$. We number the reverse lookahead symbols in the order they are matched by the DFA, which is the reverse order of the lookahead itself. The set of

Algorithm 4: Initial state set $\mathcal{I}_{\sigma_1\sigma_2}$ and $\mathcal{I}_{\max,2}$ computation for 2-character reverse lookahead

```
Input  : $\delta$, $Q$, $\Sigma$
Output: $\mathcal{I}_{\sigma_1\sigma_2}$, $\mathcal{I}_{\max,2}$
foreach $\sigma_1 \in \Sigma$ do
    foreach $\sigma_2 \in \Sigma$ do
        $\mathcal{I}_{\sigma_1\sigma_2} \leftarrow \emptyset$
        foreach $q \in Q$ do
            $\mathcal{I}_{\sigma_1\sigma_2} \leftarrow \mathcal{I}_{\sigma_1\sigma_2} \cup (\delta(\delta(q, \sigma_1), \sigma_2) \setminus \{q_e\})$
$\mathcal{I}_{\max,2} = \max_{\sigma_1,\sigma_2 \in \Sigma} \left(|\mathcal{I}_{\sigma_1\sigma_2}|\right)$
```

initial states $\mathcal{I}_{\sigma_1\ldots\sigma_k}$ constitutes the set of all states that are the target of a path through the DFA labeled by a string with postfix $\sigma_1\ldots\sigma_k$, i.e.,

$$\mathcal{I}_{\sigma_1\ldots\sigma_k} = \{s : \delta^*(x, \sigma_1\ldots\sigma_k) = s\}, \quad \forall s, x \in Q. \tag{13}$$

Let $\mathcal{I}_{\max,r}$ be the maximum number of possible initial states when using r reverse lookahead symbols (in particular, $\mathcal{I}_{\max,1} = \mathcal{I}_{\max}$). Algorithm 4 shows for a reverse lookahead of two characters how to compute initial state sets $\mathcal{I}_{\sigma_1,\sigma_2}$ and constant $\mathcal{I}_{\max,2}$. The time complexity for computing $\mathcal{I}_{\max,r}$ is $\mathcal{O}\left(|\Sigma|^r \cdot |Q| + |Q|\right)$, i.e., the algorithm is exponential in the number r of reverse lookahead symbols.

The following lemma establishes that when increasing the amount of reverse lookahead symbols, the maximum number of possible initial states $\mathcal{I}_{\max,r}$ of a DFA is bounded above by $\mathcal{I}_{\max}$.

Lemma 1 *Given a DFA, it holds that $\mathcal{I}_{max} = \mathcal{I}_{max,1} \geq \mathcal{I}_{max,2} \geq \ldots \geq \mathcal{I}_{max,\omega}$, where ω denotes the length of the longest accepting path through the DFA.*

Proof Indirect. Without loss of generality we assume a DFA with exactly one of its transitions labeled by a symbol $\sigma \in \Sigma$, and state q being the target state of this transition. For this DFA, $|\mathcal{I}_\sigma| = 1$. Given another symbol $\sigma' \in \Sigma$, we assume that $|\mathcal{I}_{\sigma'\sigma}| = 2$. Then by the definition of $\mathcal{I}_{\sigma'\sigma}$ in Eq. (13), this DFA must have two distinct states that are the target of a path labeled by a string with postfix $\sigma'\sigma$. However, this implies that these two target states have an incoming transition labeled σ, which contradicts our initial assumption that $|\mathcal{I}_\sigma| = 1$. Thus for any two symbols σ and σ', it holds that $|\mathcal{I}_\sigma| \geq |\mathcal{I}_{\sigma'\sigma}|$. The extension to the general case $|\mathcal{I}_{\sigma_1\ldots\sigma_k}| \geq |\mathcal{I}_{\sigma'\sigma_1\ldots\sigma_k}|$ is straightforward and the lemma follows. □

4.4 Time Complexity

The time complexity of sequential DFA matching is $\mathcal{O}(n)$, where n is the length of the input string. Our basic speculative DFA matching approach from Sect. 4.1 distinguishes the first chunk from subsequent chunks to partition the input string such that the matching load is balanced. The time required for parallel matching is on the order of

$$\mathcal{O}\left(\frac{n \cdot |Q|}{|Q| + |P| - 1}\right). \tag{14}$$

The speedup of Algorithm 2 over sequential matching is thus on the order of

$$\mathcal{O}\left(1 + \frac{|P| - 1}{|Q|}\right). \tag{15}$$

It follows that in terms of algorithm complexity, this approach will not produce a speed-down, i.e., it is failure-free.

Equation (16) shows the time complexity of parallel DFA matching with reduced sets of potential initial states from Sect. 4.2. Because computing $\mathcal{I}_{\max,r}$ constitutes overhead, we have an additional term $\mathcal{O}(|Q| \cdot |\Sigma|^r)$, where r is the number of reverse lookahead symbols.

$$\mathcal{O}\left(|Q| \cdot |\Sigma|^r + \frac{n \cdot |\mathcal{I}_{\max,r}|}{|\mathcal{I}_{\max,r}| + |P| - 1}\right) \tag{16}$$

If $n \gg |Q|$, or if $\mathcal{I}_{\max,r}$ is computed off-line, the additional term can be neglected. Even when computing $\mathcal{I}_{\max,r}$ on-line, for all considered cases the approach with reduced sets of potential initial states showed better performance.

Our method is capable of utilizing processors of different processing capacities, which is relevant for heterogeneous multiprocessors and for cloud computing environments. Different processor weights w encode processors' capacities. Because we employ weights to calculate chunk sizes for processors, we encode different processing capacities in the size of each processor's chunk. If we do not apply weights for processors of different processing capacities, the following equation describes the overall time complexity,

$$\mathcal{O}\left(\frac{nm}{m + p - 1}\right), \tag{17}$$

where $p = |P| \times w_{worst}$ and $w_{worst} = \min(w_0, w_1, \ldots, w_{|P|-1})$ and m is either $|Q|$ or $|\mathcal{I}_{\max,r}|$. Incorporating reduced sets of potential initial states with Eq. (15) yields a speedup on the order of

$$\mathcal{O}\left(1 + \frac{|P| - 1}{|Q| \cdot \gamma}\right), \tag{18}$$

where the ratio $\gamma = \frac{\mathcal{I}_{\max,r}}{|Q|}$ of reduced sets of potential initial states $\mathcal{I}_{\max,r}$ to $|Q|$ constitutes a structural DFA property. We have $0 < \gamma \leq 1$, where the magnitude of actual values for γ is negatively correlated to the profitability of our optimization for particular DFAs.

Table 2 Hardware specifications

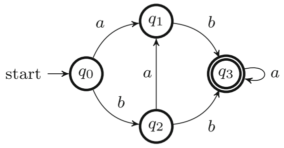
Name	CPU model	CPUs	$\frac{\text{Cores}}{\text{CPU}}$	Clock Freq.	Note
Intel MTL	Intel Xeon E7-4860	4	10	2.27 GHz	n/a
SDE emulator on local server	AVX2/Haswell on Intel Xeon E5405 host	n/a	n/a	n/a	n/a
Amazon EC2 (m2.4×large)	Intel Xeon X5550	2	4	2.67 GHz	26 EC2 CUs
Amazon EC2 (cc2.8×large)	Intel Xeon E5-2670 Sandy Bridge	2	8	2.60 GHz	88 EC2 CUs

5 Implementation

We implemented our speculative DFA matching algorithms for the three architectures summarized in Table 2. For our shared-memory multicore architecture implementation we were granted access to the Intel Academic Program Manycore Testing Lab (Intel MTL, [20]), which is an experimental environment of non-commercial, 40-core nodes provided by Intel mainly for educational purposes. POSIX threads [11] were used to parallelize DFA matching across multiple cores. To vectorize our speculative DFA matching algorithm, we employed version 2 of the Advanced Vector Extensions (AVX2) of the forthcoming Intel Haswell CPU architecture [21]. The AVX2 instruction set provides 256 bit registers enabling eightfold vectorization on 32-bit integer and single precision floating point data types. AVX2 is the first x86 instruction set extension to provide a gather-operation for vectorized indexed read operations from memory (vectorized register-indirect addressing). To the best of our knowledge, we are the first to utilize gather operations to vectorize DFA matching. Because the Haswell architecture was scheduled to be released in 2013, there was no processor with AVX2 support available at the time of writing. Hence, we used Intel's Software Development Emulator (SDE, [22]) to emulate AVX2 instructions. To evaluate our approach in a cloud computing environment, we employed m2.4×large and cc2.8×large instances of the Amazon EC2 elastic computing cloud [3]. Each EC2 instance provides a nominal dedicated compute capacity stated in Amazon's proprietary Compute Unit (CU) measure. Hardware specifications of the used Amazon EC2 instance types (nodes) are given in Table 2. For our experiments, we employed 20 instances with a total of 320 physical cores. For communication across threads, the MPI message passing interface was used.

We tailored our DFA data-structures to maximize performance and to utilize the AVX2 instruction set, in particular the novel AVX2 32-bit gather operations. To generate minimal DFAs from regular expressions, we use Grail+ [15,37], which is a formal language toolset for the manipulation and application of regular expressions and automata. Our DFA matching framework reads DFAs and input strings in Grail+ format and converts them to our framework's internal representation.

DFA transition tables are usually represented as 2-dimensional arrays, with rows for each state and one column for each character $x \in \Sigma$. With our representation,



(a)

(START)|-		0
0	a	1
0	b	2
1	b	3
2	a	1
2	b	3
3	a	3
3	-|(FINAL)	

(b)

```
SBase = { 1, 2,
          4, 3,
          1, 3,
          3, 4,
          4, 4 };
```

(c)

Str=babababbababbabababababa

`IBase={1,0,1,0,1,1,0,1,0,1,1,0,1,0,1,0,1,0,1,0};`

(d)

Fig. 8 Example DFA (**a**), Grail+ format (**b**), `SBase` 1-dimensional transition table representation (**c**), and representation of the DFA input (**d**)

2-dimensional arrays are flattened into consecutive, 1-dimensional arrays. This representation allows to store multiple DFAs of different alphabet sizes, and it facilitates application of AVX2 gather operations (i.e., gather operations allow 1-dimensional indexed reads only). Fig. 8a shows our running example DFA from Fig. 6 and the DFA's Grail+ format (Fig. 8b). Our transition table representation is given in C-like pseudo code in Fig. 8c. Grail+ encodes DFA states as integers. Lines in Grail+ format represent triples ⟨ source-state, transition-label, target-state ⟩, with the start and accepting states indicated on separate lines. Our DFA representation encodes states as row-indexes into the DFA transition table. Note that State 4 represents the error-state q_e. Row-indexes are calculated relative to the base address `SBase` of the array. In case of a second DFA stored after the running example, the second DFA's row indexes will also be stored relative to `SBase`. For the input string, we introduce a 1-dimensional array `IBase` of integers. For example, in Fig. 8d, character *a* is mapped to the value 0, and character *b* is mapped to 1. Multiple DFA input strings may be concatenated in array `IBase`. Generation of this DFA and input string representation can be trivially implemented while parsing the Grail+ DFA input data. Our representation allows to run a single DFA simultaneously on multiple input strings, or to match multiple DFAs on one or more input strings.

Listing 1 Baseline matching routine in C for a possible initial state of a chunk

```
// Get address of first and last character of chunk:
INPUT_T* curPtr=&IBase[StartPos];
INPUT_T* endPtr=&IBase[EndPos];

// Get starting state and perform matching:
STATE_T CurrentState=InitialState*NrSymbols;
for ( ; curPtr!=endPtr; curPtr++) {
 CurrentState=SBase[CurrentState + *curPtr];
}
```

Listing 1 shows how a chunk is matched for one possible initial state on multicore architectures. It should be noted that by encoding the transition table's DFA states as offsets relative to the `SBase` base address, 2-dimensional table lookups of conventional DFA representations are simplified to a 1-dimensional lookup that avoids the rows-times-column multiplication of 2-dimensional arrays—with our representation, we only add the current state's offset to the current input symbol (line 8 of Listing 1). We employ pointers to access the input and to detect loop termination, thereby avoiding the need for maintaining a separate loop counter variable. When compiled to x86-64, this matching loop consists of only two add operations, one comparison, one indexed load and one conditional jump, which compares favorable to Grail+'s matching loop implemented in C++, which requires more than an order of magnitude more instructions for the same purpose. We used a variant of Listing 1 for sequential DFA matching. This sequential matching routine was used as an efficient yardstick for the comparison to our parallelized matching algorithms, because we found our sequential matching routine to be more efficient than Grail+, and the closest approach from the related work, i.e., [19], incurred slowdowns over sequential matching (see Fig. 11 and Sect. 7).

5.1 Vectorized DFA Matching Using AVX2 Instruction Set Extensions

Listing 2 Vectorized DFA matching of chunks using AVX2 intrinsics

```
 1 int i;
 2 __m256i InpSyms, Ones = _mm256_set1_epi32 (1);
 3
 4 // Load initial indices into SBase and IBase arrays:
 5 __m256i States = _mm256_load_si256((__m256i const *)
       CStatesInit);
 6 __m256i InpIdx = _mm256_load_si256((__m256i const *)
       CInputInit);
 7
 8 for (i = ChunkLength; i>0; i--) {
 9   // Load input from IBase, indexed by InpIdx:
10   InpSyms = _mm256_i32gather_epi32 (IBase, InpIdx, 4);
11   // Calculate indices of next states:
12   States = _mm256_add_epi32 (States, InpSyms);
13   // Load next state values from SBase:
14   States = _mm256_i32gather_epi32 (SBase, States, 4);
15   // Increase input indices by one:
16   InpIdx = _mm256_add_epi32(InpIdx, Ones);
17 }
```

Listing 2 shows our core matching loop with eightfold vectorization employing AVX2 vector instruction intrinsics [21]. Data type `__m256i` represents an 8-way vector containing 8 32-bit `int` variables. Variables `States` and `InpIdx` contain the indices into the state transition table `SBase` and the input array `IBase`. They are initialized to precomputed starting-positions of chunks in lines 5 and 6. We use the `_mm256_i32gather_epi32` intrinsic to perform vectorized, indexed loads from the `SBase` and `IBase` arrays. For example, in line 8, 8 input characters are loaded from `IBase`. Note that the offsets in vector `InpIdx` are scaled by a factor of 4 (the third argument of the intrinsic), to account for the 32-bit size of type `int`.

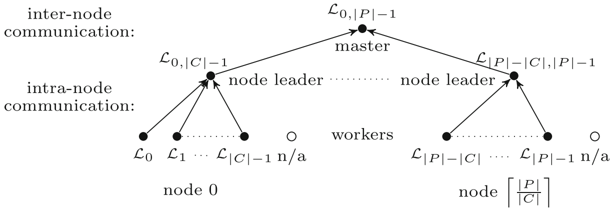


Fig. 9 Hierarchical merging of $\mathcal{L}$-vectors to reduce message delay and variability on EC2. The number of available processing cores is denoted by $|P|$, and the number of cores allocated per node is denoted by $|C|$. One core per node is left unallocated, to avoid performance degradation with hypervised EC2 nodes. Unallocated cores are denoted by symbol "$\circ$"

For further details on the used intrinsics, we refer to [21]. The reason to count the loop index variable down instead of up is because the decrement instruction will already set the x86 CPU's sign flag when we cross zero. This way we save a `cmp` instruction which yields additional 12 % of performance improvement. Neither GCC nor Intel's ICC managed to generate optimal assembly code from Listing 2, which required us to use inline assembly instead. Auto-vectorization of sequential DFA matching is out of reach for compilers, because of the dependencies between current and next DFA state.

5.2 DFA Matching on Cloud Computing Architectures

With our implementation for the EC2 cloud computing environment, we employed MPI-based message-passing to communicate between cores for merging $\mathcal{L}$-vectors. The chosen MPI implementation was MPICH2 version 1.4 [1]. As mentioned previously, parallel reduction based on binary trees did not achieve satisfactory performance. We found the message transfer times of messages between EC2 nodes too high to make binary reduction profitable. E.g., the average inter-node transfer time for a single $\mathcal{L}$-vector was 362 µs, with a standard-deviation of 3.6 %. In comparison, the same intra-node message would take on average only 2.68 µs, with a standard-deviation of 0.14 %. This observation is in line with a recent study that reports large delay variations and unstable network throughput for the EC2 cloud [46].

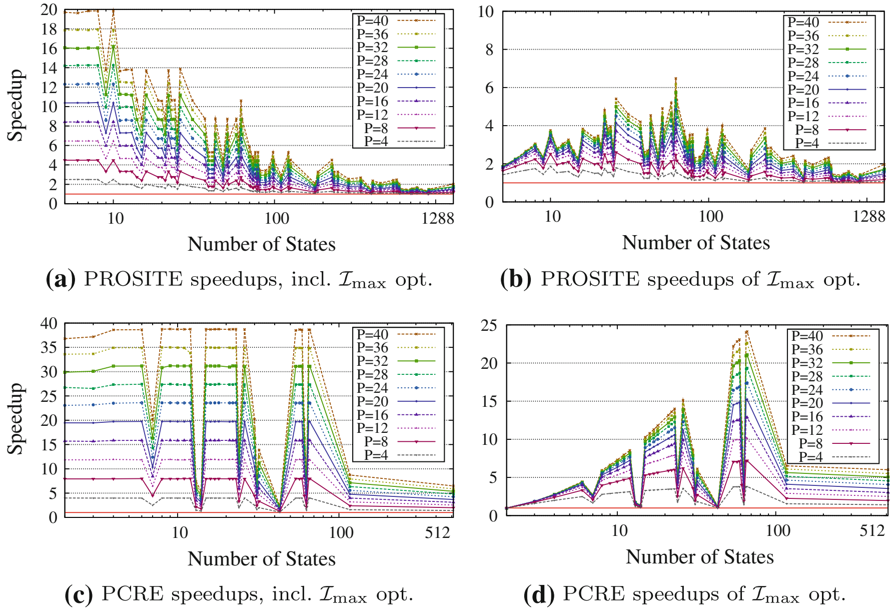
To account for the message delay and variations on EC2, we devised a variant of parallel reduction that is hierarchical with respect to intra-node and inter-node communication. This 2-tier merging approach is based on the observation that intra-node messages showed substantially lower message transfer times and variations than inter-node communication. Our reduction proceeds in two steps, as depicted in Fig. 9. In the first step, $\mathcal{L}$-vectors are merged locally by a designated node leader. In the second step, node leaders send their $\mathcal{L}$-vectors to the master process which combines them to compute the overall matching result. Without loss of generality, this 2-step merging scheme requires that on each EC2 node, DFA-matching worker processes are allocated to adjacent chunks. Our worker-to-node allocation scheme is parameterized by the number of cores to utilize per node, denoted by $|C|$. For reasons explained below, we leave one core unallocated per EC2 node. Figure 9 depicts the computation of $\mathcal{L}$-vectors by workers (for one chunk), node leaders (the combined map over all chunks matched on a node) and the master (the overall map from the first to the last chunk). Unallocated cores are denoted by symbol "$\circ$".

Our two-tier merging scheme outperformed parallel binary reduction and sequential merging for even the largest EC2 clusters (i.e., up to 20 nodes, which is the maximum possible EC2 cluster size [4]). We found MPI messages among processes on the same node to show both low latency and low variability. We conjecture that MPICH2 applies shared-memory message passing optimizations similar to [24] for node-local communication. Moreover, node-local communication is free from delay variations induced by the network that connects nodes. Therefore, with our merging scheme the only communication step subjected to EC2's message variability is the merging step conducted by the master. This compares favorably to any parallel reduction scheme with more than one reduction step involving inter-node communication, because each such reduction step may suffer from message delays caused by the underlying network.

As mentioned above, we deliberately left one core per EC2 node unallocated. We observed that without sacrificing one core per EC2 node, there was a high probability that one of the workers on each node would experience a matching performance on the order of one magnitude lower than the workers on the remaining cores. This performance degradation did not affect the offline profiling step, for which we took the median of a series of partial matching runs. However, this performance degradation randomly showed with DFA matching. Because we could not reproduce this problem on a local cluster of Linux computers, we attribute this performance degradation to EC2 hypervisor activities that occasionally preempted the execution of one arbitrary worker thread per node. Leaving one core unallocated on EC2 eliminated this problem. Given the increasing numbers of cores per CPU, leaving one core unallocated can be considered an increasingly small sacrifice (e.g., our experiments were conducted with EC2 nodes providing 8 and 16 cores, respectively).

6 Experimental Results

We conducted experiments for both our basic and optimized speculative matching algorithms and the approach of Holub and Štekr presented in [19]. We employed 299 regular expressions from the PCRE library [34] and 110 protein patterns from the PROSITE protein database [35]. Protein patterns were selected as an example for the application domain of DNA sequence analysis. We compared our algorithms to the baseline sequential DFA matching algorithm from Sect. 5 and to the currently used matching engine that comes with PROSITE. All PCRE regular expressions and PROSITE protein patterns were converted to unique minimum DFAs using Grail+ [15, 37]. All experiments except the experiments on EC2 were conducted with inputs of one million characters. Because we employed up to 288 cores on EC2, the problem sizes of one million characters turned out too small for precise performance measurements. Thus we used inputs of 8 million characters on EC2. We show the scalability of our approach on both the MTL and the EC2 platforms for up to 10 billion characters in Sect. 6.3. Note that for increased readability we represent speed-downs by negative values instead of fractional values. For example, conventional denotation for a 2× speed-down is $\frac{1}{2}$ but we use -2.



(a) PROSITE speedups, incl. $\mathcal{I}_{\max}$ opt.

(b) PROSITE speedups of $\mathcal{I}_{\max}$ opt.

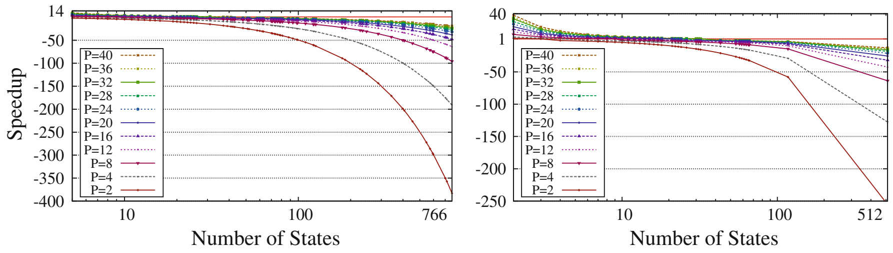
(c) PCRE speedups, incl. $\mathcal{I}_{\max}$ opt.

(d) PCRE speedups of $\mathcal{I}_{\max}$ opt.

Fig. 10 Speedups on a shared-memory 4 CPU (40 cores) node of the Intel MTL for PROSITE patterns (**a**) and PCRE (**c**), including the $\mathcal{I}_{\max}$ optimization for four symbols reverse lookahead. **b** and **d** show speedups achieved by the $\mathcal{I}_{\max}$ optimization over matching for $|Q|$

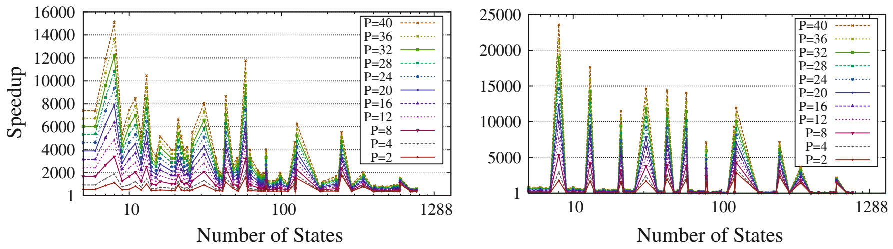
Figure 10 shows the results of our speculative parallel DFA membership test with and without applying four symbol reverse lookahead, for the PROSITE and PCRE benchmark suites conducted on the Intel MTL. We used GCC 4.5.1 on RedHat RHEL 5.4 (x86_64 kernel version 2.6.18-164.el5). x-axes denote the number of states $|Q|$, and y-axes denote the speedup over sequential matching. We note the following observations: (1) Our algorithms always show better performance than sequential matching, despite the overhead from redundant computations incurred by speculative parallelization. Redundant computations constitute matching of subsequent chunks for multiple DFA states, which contrasts sequential DFA matching where the input is only matched for the start state q_0. (The red horizontal lines denote the break-even point where the speedup over sequential matching is 1). The fact that there are no speed-downs validates the failure-freedom of our speculative parallelization. (2) Speedups are always proportional to $|P|$, as predicted by the complexity analysis in Sect. 4.4. This proves our basic assumption that the number of symbols to be processed per processor decides the overall matching time despite the overhead due to parallelization. The performance improvements due to our $\mathcal{I}_{\max}$ optimization are shown in Fig. 10b, d.

This result compares favorable to an approach presented in [19], which has a complexity of $\mathcal{O}(\frac{n \cdot |Q|}{|P|})$ and thus achieves speedups only if the number of processors is larger than the number of states. We evaluated the approach from [19] for both PCRE and the PROSITE patterns, as depicted in Fig. 11. In-line with the algorithm's complexity results, the previous approach cannot achieve speedups when $|P| \leq |Q|$. We



(a) PROSITE protein patterns **(b)** PCRE regular expressions

Fig. 11 Performance of the approach from [19] on the Intel MTL



(a) Speedup over ScanProsite **(b)** Speedup over `grep`

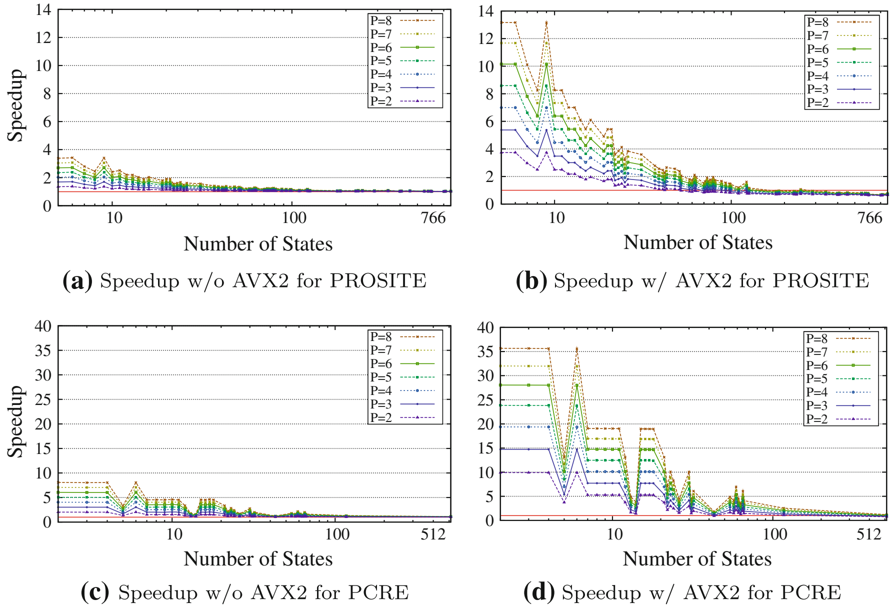
Fig. 12 Performance of our approach compared to ScanProsite (**a**) and the UNIX `grep` utility (**b**)

observed an almost 390× speed-down for a DFA with 788 states. In contrast, our algorithm achieved a speedup between 2.3× and 38.8× for PCRE, and between 1.3× and 19.9× for PROSITE.

Another experiment conducted on the MTL is the comparison to ScanProsite [14,39], which is the reference implementation from the PROSITE protein database. ScanProsite is used to detect signature matches in protein sequences. The tool is implemented in Perl; it can be used to find all substrings that match a certain PROSITE pattern. We parameterized ScanProsite to find only one match to compare with our optimized DFA matching algorithm which determines whether an input string contains a certain pattern or not. For a second comparison, we employed the UNIX grep utility with ScanProsite. Grep constructs a DFA and uses the Boyer-Moore algorithm for matching [17]; it is faster than Perl which uses backtracking [13]. As shown in Fig. 12, our algorithm using four symbol reverse lookahead is 559.3–15079.7 times faster than ScanProsite, and 62.1–23572.0 times faster than the UNIX `grep` utility.

6.1 Performance of Vectorized DFA Matching Using AVX2 Instruction Set Extensions

At the time of writing, CPUs supporting AVX2 instruction set extensions were not commercially available. To validate our vectorized code, and to get an indication on



(a) Speedup w/o AVX2 for PROSITE

(b) Speedup w/ AVX2 for PROSITE

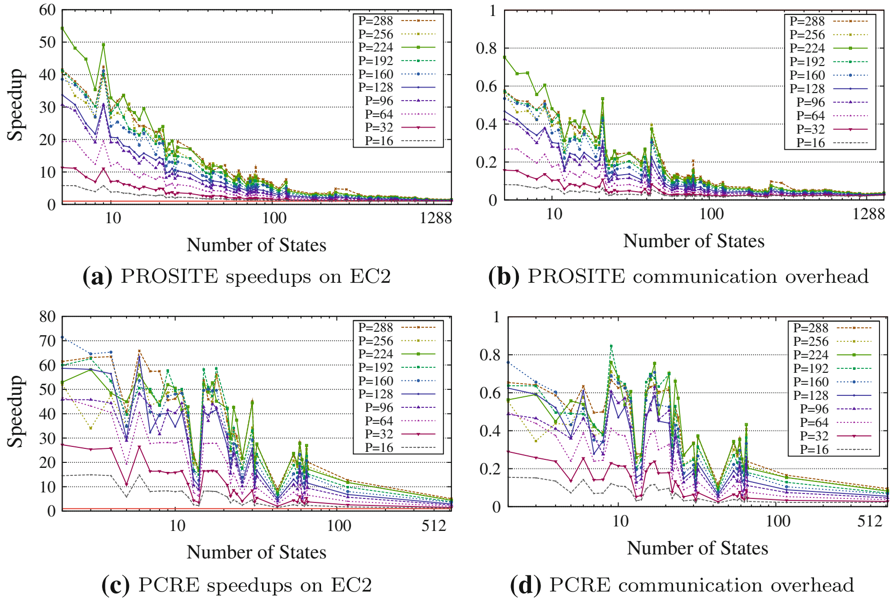
(c) Speedup w/o AVX2 for PCRE

(d) Speedup w/ AVX2 for PCRE

Fig. 13 Speedups from AVX2 eightfold vectorization for PROSITE patterns (**b**) and PCRE (**d**). **a** and **c** depict the achieved speedups without vectorization

the speedups obtainable with AVX2, we resorted to Intel's SDE emulator [22], version 4.46.0. SDE is not cycle-accurate, but it provides the number of machine instructions "executed" by the emulated binary. We used the number of executed machine instructions as the basis of our performance comparison of scalar and vectorized code. *Speedup* throughout Sect. 6.1 thus denotes a ratio of executed machine instructions, as opposed to observed execution time. Performance on real hardware, e.g., Intel's Haswell microarchitecture, must be expected to vary to the extent of variations in the cycles per instruction (CPI) between scalar and vectorized code. For compilation of code with AVX2 intrinsics, we used ICC version 12.1.4.

Figure 13 compares the speedups of scalar and vectorized DFA membership tests applying the $\mathcal{I}_{\max}$ optimizations for one symbol lookahead. Eightfold vectorization using AVX2 instructions achieved a 4.45× improvement over scalar code. Furthermore, we observed that the expected speedups on an emulated 8-core machine with AVX2 are on the order of magnitude of a 40-core node of the MTL. The speedups range from 1.2× to 35.7× for PCRE and 0.7× to 13.2× for PROSITE. Speedup is again proportional to $|P|$, showing that vectorization is in-line with our complexity analysis from Sect. 4.4. We observed a 16.0 % speed-down on average (maximum 31.5 %) with very large DFAs due to the overhead of our parallelization for SIMD operations. This speed-down is not innate to the algorithms, but due to our implementation, in particular the way chunks are allocated to SIMD vector units. The speed-down can be overcome by increasing the problem size (which we refrained from, to keep experiments consistent).



(a) PROSITE speedups on EC2

(b) PROSITE communication overhead

(c) PCRE speedups on EC2

(d) PCRE communication overhead

Fig. 14 Performance on cloud computers (cc2.8×large instance type on EC2) with PROSITE patterns (**a**) and PCRE (**c**). **b** and **d** show proportional overhead of MPI communication cost with PROSITE patterns and PCRE respectively

6.2 DFA Matching Performance on Cloud Computing Architectures

We conducted experiments on the Amazon EC2 elastic computing cloud to determine the performance of our speculative DFA matching algorithms on distributed-memory architectures, employing up to 20 nodes and 288 cores. We explored the adaptation of our load-balancing approach to EC2 nodes of varying processing capacities. For the convenience of operating a cluster of EC2 nodes, we used StarCluster [44] version 0.93.3, an open source cluster-computing toolkit for EC2.

Experiments were conducted on up to 20 cc2.8×large EC2 instances, which provide 16 cores per node. We again employed four symbols reverse lookahead with our approach. For reasons discussed in Sect. 5.2, we occupied 15 out of 16 cores, resulting in 300 cores in total. For better presentation, Fig. 14 shows our experimental results with cluster sizes that are a multiple of 32 cores. We used version 1.4 of the MPICH2 MPI implementation [1], which provided higher performance on EC2 than OpenMPI [32] version 1.4.3.

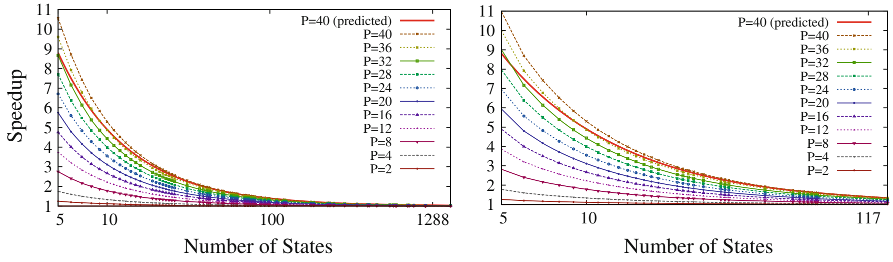
We found the communication costs between nodes an important factor on the EC2 cloud. We instrumented our matching framework to determine the communication overhead. Figure 14a, c show speedups including communication costs, and Fig. 14b, d depict the ratio of time spent for communication to overall execution time. Although graphs shown in Fig. 14 are irregular due to the instability of the EC2 network, we can observe that the communication costs increase as the number of processors grows. The communication cost decreases as $|Q|$ grows, which follows from the fact that

Table 3 Effectiveness of the load-balancing scheme on six configurations of inhomogeneous clusters consisting of two types of Amazon EC2 instances, m2.4×large and cc2.8×large

EC2 instances		PROSITE			PCRE		
Fast	Slow	Min.	Avg.	Max.	Min.	Avg.	Max.
0	5	0.0036	0.0102	0.0298	0.0046	0.0149	0.0696
1	4	0.0031	0.0086	0.0360	0.0036	0.0108	0.0355
2	3	0.0033	0.0090	0.0275	0.0062	0.0121	0.0427
3	2	0.0051	0.0116	0.0248	0.0083	0.0186	0.0707
4	1	0.0060	0.0130	0.0700	0.0093	0.0194	0.0707
5	0	0.0056	0.0119	0.0305	0.0095	0.0188	0.0412

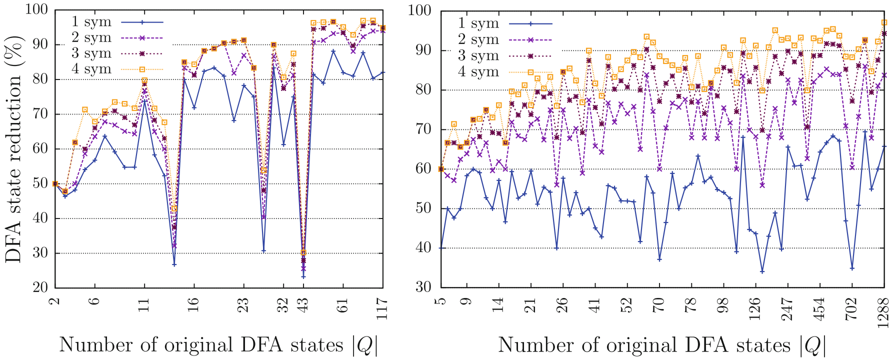
the required matching time increases with $|Q|$, which de-emphasizes communication costs. This observation explains why PCRE benchmarks, which show smaller $\mathcal{I}_{\max}$ constants and smaller sets of possible initial states, are more impacted by communication overhead than PROSITE benchmarks.

The goal of our load-balancing mechanism is to determine chunk sizes such that all processing cores are utilized equally, i.e., take equally long for matching their assigned chunk. Processor capacities are incorporated in the form of weights (see Eq. (4)). To evaluate the load-balance achieved with our speculative DFA matching computations, we used two different types of Amazon EC2 instances, namely cc2.8×large (denoted as “Fast” in Table 3) and m2.4×large (denoted as “Slow” in the second column of the table). Although the clock frequencies of these EC2 instance types do not differ much (see Table 2), the difference of the processor capacities is observable. We found the ratio of actual processing capacity of cc2.8×large compared to m2.4×large to be 1.41 on average, meaning that cc2.8×large on average computes 41 % faster than m2.4×large. For this experiment, we allocated inhomogeneous clusters consisting of various numbers of cc2.8×large and m2.4×large instances. To get an indication for the effectiveness of our load-balancing scheme, we determined the standard-deviations of DFA matching times across all cores of such inhomogeneous EC2 clusters. A balanced load would then be indicated by standard deviations close to zero. E.g., the experiment from row 5 was conducted on a mix of four cc2.8×large instances and one m2.4×large instance. The maximum observed standard deviation of execution times was 7.0 %, with 0.6 % minimum standard deviation and 1.3 % average across the PROSITE benchmark suite. During experiments, we noticed that capacities of cluster nodes could change slightly across cluster invocations, making the re-estimation of processor capacities necessary at cluster startup time. (This is in line with the findings from [41], on performance unpredictability of cloud computing environments). Hence the adaptability of our load-balancing scheme with respect to processor capacities is essential on cloud computing environments. Our observed proportional standard deviations of execution times are very low, around 1 % on average, as shown in Table 3. In particular, the presented load-balancing scheme adapts well to different configurations of inhomogeneous clusters.



(a) PROSITE patterns **(b)** PCRE

Fig. 15 Speedups on the Intel MTL node over sequential matching, without $\mathcal{I}_{\max}$ optimization. Observed speedups closely follow the prediction from Eq. (15). The predicted speed-up for 40 cores is shown by the graph labelled "P = 40 (predicted)"



(a) PCRE **(b)** PROSITE

Fig. 16 Original sizes of DFAs and the reduction rate for various numbers of reverse lookahead symbols: $\mathcal{I}_{\max,1}$, $\mathcal{I}_{\max,2}$, $\mathcal{I}_{\max,3}$ and $\mathcal{I}_{\max,4}$

6.3 Performance Impact of Structural DFA Properties and Scalability to Large Input Sizes

If we regard the number of processing cores as a constant, then the expected speedup of the proposed speculative DFA matching approach solely depends on the number of DFA states $|Q|$ (see Eq. (15)). Figure 15 shows the speedups that we obtain without the $\mathcal{I}_{\max}$ optimization. The line labeled "Theoretical Speedup for P = 40" depicts the expected speedup according to Eq. (15). The observed speedups closely follow this trend. Speedups without $\mathcal{I}_{\max}$ optimization are thus very regular. This contrasts the speedups obtained with $\mathcal{I}_{\max}$ optimization enabled, because DFAs with the same number of states $|Q|$ can vary drastically in $\mathcal{I}_{\max}$, i.e., their maximum number of possible initial states. As a consequence, the number of DFA states $|Q|$ cannot predict the performance of the $\mathcal{I}_{\max}$ optimization. We will thus consider to what extent different numbers of reverse lookahead symbols reduce the number of possible initial start states.

We investigated the sizes of possible initial state sets for the PCRE and PROSITE benchmark suites for 1, 2, 3 and 4 reverse lookahead symbols. Figure 16 depicts the

Table 4 Average size of $\mathcal{I}_{\max,r}$ compared to $|Q|$, for r reverse lookahead symbols

r	0 (%)	1 (%)	2 (%)	3 (%)	4 (%)
PCRE	100	33.7	26.4	23.7	21.7
PROSITE	100	47.2	29.2	20.5	16.0

number of states $|Q|$ and the initial state reduction rates for 299 PCRE benchmark DFAs and 110 PROSITE protein patterns. (For DFAs with the same number of states, the possible initial state set sizes were averaged). For example, the rightmost, largest DFA in Fig. 16b consists of $|Q|$= 1,288 states. One-symbol reverse lookahead eliminates 65 % of $|Q|$. Two-symbol, three-symbol and four-symbol lookahead remove 83, 94 and 97 % of all states. It follows from Fig. 16 that DFAs exhibit variations in their initial state reduction rate, which has an impact on matching performance. For example, the PCRE DFA with 43 states (Fig. 16a) shows $\mathcal{I}_{\max}$ reduction rates below 31 %. The resulting impact on performance can be observed in Fig. 10c with the Intel MTL node and in Fig. 14c for the EC2 cloud.

The average size of possible initial state sets for 1, 2, 3 and 4 reverse lookahead symbols compared to the overall number of states $|Q|$ is depicted in Table 4. Applying a reverse lookahead of one symbol to the PCRE benchmarks reduces the number of possible initial states on average to 33.7 % of the original states. Applying 2, 3 and 4 reverse lookahead symbols yielded further reductions of 7, 10 and 12 % over $|Q|$. With the PROSITE benchmarks, one symbol reverse lookahead reduced on average to 47.2 % of the original states. Applying 2, 3 and 4 reverse lookahead symbols yielded further reductions of 18, 26 and 31 % over $|Q|$. The profitability of reverse lookahead is a static property of DFAs, which is reflected in this data: while for PCRE one symbol lookahead already yields a large reduction on the number of states, lookahead >2 symbols does not provide substantial improvement. However, with PROSITE, one symbol reverse lookahead provided a smaller improvement, while reverse lookahead up to 4 symbols yielded steady gains.

Because of the exponential time complexity to compute $\mathcal{I}_{\max,r}$, there is a trade-off between the overhead of the reverse lookahead computation and the obtainable performance gains. To quantify this overhead, we investigated the cost of reverse lookahead computations on an Intel Xeon 5120 CPU. Figure 17a shows the overhead in microseconds to compute $\mathcal{I}_{\max,r}$ for an example DFA of $|Q| = 5$ up to three

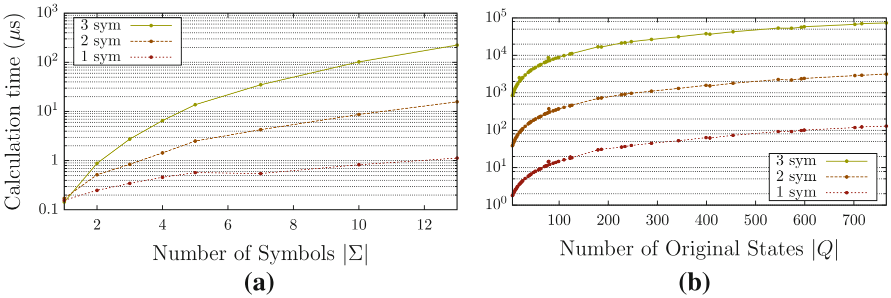


Fig. 17 Required overhead due to $\mathcal{I}_{\max,r}$ calculation over $|\Sigma|$ (**a**) and $|Q|$ (**b**)

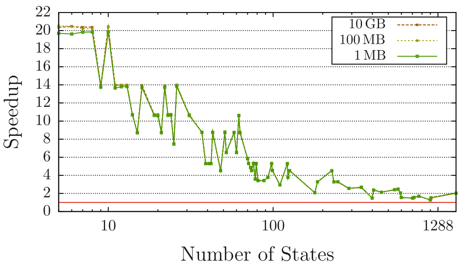
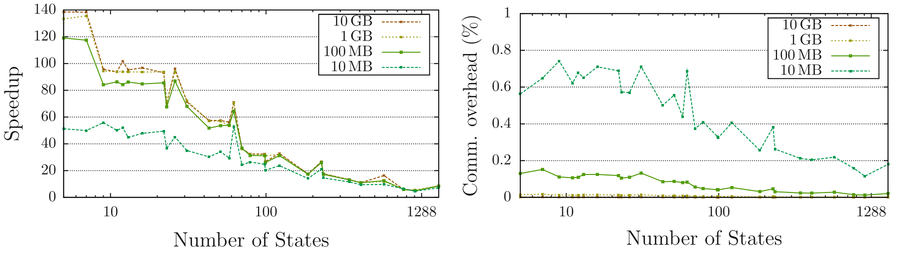


Fig. 18 Speedups for varying input sizes and P = 40 cores on the Intel MTL node



(a) PROSITE speedup

(b) PROSITE communication overhead

Fig. 19 Performance variation (**a**) over different sizes of inputs and its proportional communication overhead (**b**) with PROSITE patterns on cloud computers (cc2.8×large instance type on EC2)

reverse lookahead characters. As expected, the overhead is exponential in the size of Σ. Figure 17b depicts the overhead for increasing numbers of states. Because $\mathcal{I}_{\max,r}$ is a static property of a DFA, it can be computed off-line, and then loaded when the matching operation is performed. This way the overhead can be avoided with DFAs that are matched many times (e.g., protein patterns from databases).

We have experimentally evaluated to what extent the size of the input string affects the DFA matching performance of our approach. The order of magnitude of the speedups from Eq. (15) does not contain the size of the input (n), which is reflected by the performance data obtained on the Intel MTL node: for input sizes of 1 MB, 100 MB and 10 GB, the obtained speedups are almost identical (Fig. 18). Our algorithms thus scale well with respect to large input sizes on the MTL shared memory multicore architecture.

Figure 19a depicts the performance for input sizes of 10 MB, 100 MB, 1 GB and 10 GB for the PROSITE patterns executed on EC2 for 288 cores. Unlike shared memory architectures, our DFA matching algorithm achieves higher performance for long (10 GB) input. This is due to the communication costs, which depend on the DFA size (for transmitting $\mathcal{L}$-vectors with the final state reduction), but which are independent of the DFA input size. Larger DFA inputs incur longer overall chunk matching times, which de-emphasize the high (but constant) communication costs for a given DFA. Figure 19b depicts the proportional communication costs that we measured for varying input sizes. It follows that for 1 and 10 GB input sizes, the proportion of communication overhead with respect to the overall execution time is close to zero. However, with input sizes of 100 MB and especially 10 MB, the time spent for communication constitutes a large part of the overall execution time. As shown in Fig. 19a, our DFA matching approach scaled well for input-sizes of up to 10 GB on the EC2 computing cloud.

For sake of completeness, we state the execution time costs for computing DFAs from PROSITE protein patterns. We used Grail+ to create nondeterministic finite automata (NFAs) from regular expressions, convert NFAs to DFAs, and minimize DFAs. We did not apply parallel versions of algorithms for DFA creation and minimization [12,45]. On average, it took 8 min 21.5 s to convert a PROSITE pattern to a minimal DFA, and 4 min 35.8 s to convert a pattern to a non-minimal DFA. Our speculative DFA matching approach is suitable to both minimal and non-minimal DFAs.

7 Related Work

Locating a string in a larger text has applications with text editing, compiler front-ends and web browsers, Internet search engines, computer security, and DNA sequence analysis. Early string searching algorithms such as Aho–Corasick [2], Boyer–Moore [9] and Rabin–Karp [25] efficiently match a finite set of input strings against an input text.

Regular expressions allow the specification of infinite sets of input strings. Converting a regular expression to a DFA for DFA membership tests is a standard technique to perform regular expression matching. The specification of virus signatures in intrusion prevention systems [10,38,43] and the specification of DNA sequences [8,42] constitute recent applications of regular expression matching with DFAs.

Considerable research effort has been spent on parallel algorithms for DFA membership tests. Ladner et al. [26] applied the parallel prefix computation for DFA membership tests with Mealy machines. Hillis and Steele [18] applied parallel prefix computations for DFA membership tests on the 65,536 processor Connection Machine. Ravikumar's survey [36] shows how DFA membership tests can be stated as a chained product of matrices. Because of the underlying parallel prefix computation, all three approaches perform a DFA membership test on input size n in $\mathcal{O}(\log(n))$ steps, requiring n processors. Their algorithms handle arbitrary regular expressions, but the underlying assumption of a massive number of available processors can hardly be met in most practical settings. Misra [31] derived another $\mathcal{O}(\log(n))$ string matching algorithm. The number of required processors is on the order of the product of the two string lengths and hence not practical.

A straight-forward way to exploit parallelism with DFA membership tests is to run a single DFA on multiple input streams in parallel, or to run multiple DFAs in parallel. This approach has been taken by Scarpazza et al. [40] with a DFA-based string matching system for network security on the IBM Cell BE processor. Similarly, Wang et al. [47] investigated parallel architectures for packet inspection based on DFAs. Both approaches assume multiple input streams and a vast number of patterns (i.e., virus signatures), which is common with network security applications. However, neither approach parallelizes the DFA membership algorithm itself, which is required to improve applications with single, long-running membership tests such as DNA sequence analysis.

Scarpazza et al. [40] utilize the SIMD units of the Cell BE's synergistic processing units to match multiple input streams in parallel. However, their vectorized DFA matching algorithm contains several SISD instructions and the reported speedup from 16-way vectorization is only a factor of 2.51. In contrast, our proposed 8-way vectorized DFA membership test avoids SISD instructions, achieving a speedup of 4.45 over the sequential version.

Recent research efforts focused on speculative computations to parallelize DFA membership tests. Holub and Štekr [19] were the first to split the input string into chunks and distribute chunks among available processors. Their speculation introduces a substantial amount of redundant computation, which restricts the obtainable speedup for general DFAs to $\mathcal{O}(\frac{|P|}{|Q|})$, where $|P|$ is the number of processors, and $|Q|$ is the number of DFA states. Their algorithm degenerates to a speed-down when $|Q|$ exceeds the number of processors (see also Sect. 6, Fig. 11). To overcome this problem, Holub and Štekr specialized their algorithm for k-local DFAs. A DFA is k-local if for every word of length k and for all states $p, q \in Q$ it holds that $\delta^*(p, w) = \delta^*(q, w)$. Starting the matching operation k symbols ahead of a given chunk will synchronize the DFA into the correct initial state by the time matching reaches the beginning of the chunk, which eliminates all speculative computation. Holub and Štekr achieve a linear speedup of $\mathcal{O}(|P|)$ for k-local automata. Unlike Holub and Štekr's approach, our DFA parallelization avoids speed-downs altogether. We use structural properties of general DFAs to limit the amount of speculation. In particular, the restriction to k-local automata is not required. We have vectorized our speculative matching routine, and we have extensively evaluated our approach on a 40-core shared memory architecture, for AVX2 vector instructions, and on the Amazon EC2 cloud infrastructure.

Jones et al. [23] reported that with the IE 8 and Firefox web browsers 3–40 % of the execution-time is spent parsing HTML documents. To speed up browsing, Jones et al. employ speculation to parallelize token detection (lexing) of HTML language front-ends. Similar to Holub and Štekr's k-local automata, they use the preceding k characters of a chunk to synchronize a DFA to a particular state. Unlike k-locality, which is a static DFA property, Jones et al. speculate the DFA to be in a particular, frequently occurring DFA state at the beginning of a chunk. Speculation fails if the DFA turns out to be in a different state, in which case the chunk needs to be re-matched. Lexing HTML documents results in frequent matches, and the structure of regular expressions is reported to be simpler than, e.g., virus signatures [29]. Speculation is facilitated by the fact that the state at the beginning of a token is always the same, regardless where lexing started. A prototype implementation is reported to scale up to six of the eight synergistic processing units of the Cell BE.

The speculative parallel pattern matching (SPPM) approach by Luchaup et al. [28, 29] uses speculation to match the increasing network line-speeds faced by intrusion prevention systems. SPPM DFAs represent virus signatures. Like Jones et al., DFAs are speculated to be in a particular, frequently occurring DFA state at the beginning of a chunk. SPPM starts the speculative matching at the beginning of each chunk. With every input character, a speculative matching process stores the encountered DFA state for subsequent reference. Speculation fails if the DFA turns out to be in a different state at the beginning of a speculatively matched chunk. In this case re-matching continues until the DFA synchronizes with the saved history state (in the worst case, the whole chunk needs to be re-matched). A single-threaded SPPM version is proposed to improve performance by issuing multiple independent memory accesses in parallel. Such pipelining (or interleaving) of DFA matches is orthogonal to our approach, which focuses on latency rather than throughput.

SPPM assumes all regular expressions to be suffix-closed, which is the common scenario with intrusion prevention systems; A regular expression is suffix-closed if matching a given string w implies that w followed by any suffix is matched, too. A suffix-closed regular language has the property that $x \in L \Leftrightarrow \forall w \in \Sigma^* : xw \in L$.

Unlike SPPM and the approach by Jones et al., our speculative DFA matching approach does not rely on a heavily biased distribution of DFA state frequencies. Instead, we use static DFA properties to minimize speculative matching overhead. Our approach is not restricted to suffix-closed regular expressions, and our speculation does not rely on the common case being a match (Jones et al.), or the common case being a non-match (SPPM). To the best of our knowledge, we are the first to employ SIMD gather-operations to fully vectorized the DFA matching process. Our DFA membership test provides a load-balancing mechanism for clusters and cloud computing environments. Unlike previous approaches, our speculative matching algorithm cannot result in a speed-down. We conducted an extensive experimental evaluation on a 40-core shared memory architecture, on a simulator for AVX2 vector instructions, and on the Amazon EC2 cloud infrastructure. Our benchmarks consist of 299 regular expressions from the PCRE library [34], and of 110 patterns from the PROSITE protein pattern database [42]. We analyzed the complexity of our speculative matching algorithm, and we provide insight on achievable scalability on shared-memory and cloud-computing environments. This paper is the extended, journal version of an informal one-page abstract presented at the 4th annual meeting of the Asian Association for Algorithms and Computation [6], and a preliminary technical report [7].

8 Conclusions

We have presented a speculative DFA pattern matching method for shared-memory, SIMD and cloud computing environments. Our parallel matching algorithm exploits structural DFA properties to minimize the speculative overhead. To the best of our knowledge, this is the first speculative DFA matching approach that is *failure-free*, i.e., (1) it maintains sequential semantics, and (2) it avoids speed-downs altogether. On architectures with a SIMD gather-operation for indexed memory loads, our matching operation is fully vectorized. Communication patterns specifically for the characteristics of cloud computing environments are provided. The proposed load-balancing scheme uses an off-line profiling step to determine the matching capacity of each participating processor. Based on matching capacities, DFA matches are load-balanced on inhomogeneous parallel architectures. We have shown that our algorithms have a better time complexity than previous work. We conducted an extensive experimental evaluation of PCRE and PROSITE benchmarks on a 4 CPU (40 cores) shared-memory node of the Intel Academic Program Manycore Testing Lab (Intel MTL), on the Intel AVX2 SDE simulator for 8-way fully vectorized SIMD execution, and on a 20-node (288 cores) cluster of the Amazon EC2 computing cloud. We showed the scalability of our approach for DFAs of up to 1,288 states, and input-strings of up to 10 GB. Our results predict that speculative parallel DFA matching can produce substantial speedups. Unlike previous methods, our technique does not impose any restriction on the matched regular expressions.

Acknowledgments Research partially supported by the National Research Foundation of Korea (NRF) grants funded by the Korean government (MEST) (Grant No. 2010-0005234, 2012R1A1A2044562 and 2012K2A1A9054713), through the Global Ph.D. Fellowship Program 2011 of the NRF (Grant No. 2010-0008582), and by the Intel Academic Program Manycore Testing Lab.

References

1. A high-performance and widely portable implementation of the MPI standard (MPICH2) Web Site: http://www.mcs.anl.gov/research/projects/mpich2. Retrieved Aug 2012. MPICH2 version 1.4
2. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun. ACM **18**(6), 333–340 (1975)
3. Amazon Web Services: EC2 web site. http://aws.amazon.com/ec2. Retrieved Aug 2012
4. Amazon Web Services: EC2 FAQs. http://aws.amazon.com/ec2/faqs. Retrieved Jan 2013
5. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Zaharia, M.: Above the clouds: a Berkeley view of cloud computing. Technical report, University of California at Berkeley, Electrical Engineering and Computer Sciences (2009)
6. Asian Association for Algorithms and Computation (AAAC): 4th annual meeting website (2011). http://3glab.cs.nthu.edu.tw/aaac2011. Retrieved Jan 2013
7. Burgstaller, B., Han, Y.-S., Jung, M., Ko, Y.: On the parallelization of DFA membership tests. Technical Report. TR-0003, Department of Computer Science, Yonsei University, Seoul 120–749, Korea. http://elc.yonsei.ac.kr/PDFA.html (2011)
8. Boeckmann, B., Bairoch, A., Apweiler, R., Blatter, M., Estreicher, A., Gasteiger, E., Martin, M., Michoud, K., O'Donovan, C., Phan, I., et al.: The SWISS-PROT protein knowledgebase. Nucleic Acids Res. **31**(1), 365 (2003)
9. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Commun. ACM **20**(10), 762–772 (1977)
10. Brumley, D., Newsome, J., Song, D., Wang, H., Jha, S.: Towards automatic generation of vulnerability-based signatures. In: Proceedings of the 2006 IEEE Symposium on Security and Privacy, SP '06, pp. 2–16. IEEE Computer Society (2006). doi:10.1109/SP.2006.41
11. Butenhof, D.R.: Programming with POSIX Threads. Addison-Wesley Longman Publishing Co., Inc., Boston, MA (1997)
12. Choi, H., Burgstaller, B.: Non-blocking parallel subset construction on shared-memory multicore architectures. In: Proceedings of the 11th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2013). CRPIT (2013)
13. Cox, R.: Regular expression matching can be simple and fast (but is slow in Java, Perl, PHP, Python, Ruby,...) (2007). http://swtch.com/rsc/regexp/regexp1.html
14. Gattiker, A., Gasteiger, E., Bairoch, A.: ScanProsite: a reference implementation of a PROSITE scanning tool. Appl. Bioinform 1(2), 107–108 (2002)
15. Grail+ Project Web Site: http://www.csd.uwo.ca/Research/grail. Retrieved Aug 2012
16. Gschwind, M., Hofstee, H.P., Flachs, B., Hopkins, M., Watanabe, Y., Yamazaki, T.: Synergistic Processing in Cell's Multicore Architecture. IEEE Micro 26(2), 10–24 (2006). doi:10.1109/MM.2006.41
17. Haertel, M.: Why GNU grep is fast (2010). http://lists.freebsd.org/pipermail/freebsd-current/2010-August/019310.html
18. Hillis, W.D., Steele Jr, G.L.: Data parallel algorithms. Commun. ACM **29**(12), 1170–1183 (1986). doi:10.1145/7902.7903
19. Holub, J., Štekr, S.: On parallel implementations of deterministic finite automata. In: Proceedings of the 14th International Conference on Implementation and Application of Automata, pp. 54–64 (2009)
20. Intel Academic Program Manycore Testing Lab Site: http://software.intel.com/en-us/articles/intel-many-core-testing-lab. Retrieved Aug 2012
21. Intel Advanced Vector Extensions Programming Reference: http://software.intel.com/en-us/avx. Retrieved Aug 2012. JUNE 2011 version

22. Intel Software Development Emulator (SDE) Web Site: http://software.intel.com/en-us/articles/intel-software-development-emulator. Retrieved Aug 2012. SDE version 4.46.0
23. Jones, C.G., Liu, R., Meyerovich, L., Asanović, K., Bodík, R.: Parallelizing the web browser. In: Proceedings of the First USENIX Conference on Hot Topics in Parallelism, HotPar'09, pp. 7–7. USENIX Association, Berkeley, CA, USA (2009)
24. Karonis, N., Supinski, B.R.D., Foster, I., Gropp, W., Lusk, E., Bresnahan, J.: Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In: Proceedings of the 14th International Symposium on Parallel and Distributed Processing, IPDPS '00, pp. 377. IEEE Computer Society, Washington, DC, USA (2000)
25. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. **31**(2), 249–260 (1987)
26. Ladner, R.E., Fischer, M.J.: Parallel prefix computation. J. ACM **27**(4), 831–838 (1980). doi:10.1145/322217.322232
27. Lin, C., Snyder, L.: Principles of Parallel Programming. Addison Wesley, Boston (2008)
28. Luchaup, D., Smith, R., Estan, C., Jha, S.: Multi-byte regular expression matching with speculation. In: Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection, RAID '09, pp. 284–303. Springer, Berlin, Heidelberg (2009). doi:10.1007/978-3-642-04342-0_15
29. Luchaup, D., Smith, R., Estan, C., Jha, S.: Speculative parallel pattern matching. IEEE Trans. Inf. Forensics Secur. **6**(2), 438–451 (2011)
30. Luján, M., Gustafson, P., Paleczny, M., Vick, C.A.: Speculative parallelization–eliminating the overhead of failure. In: Perrott, R.H., Chapman, B.M., Subhlok, J., de Mello, R.F., Yang, L.T. (eds.) HPCC, Lecture Notes in Computer Science, vol. 4782, pp. 460–471. Springer, Berlin (2007)
31. Misra, J.: Derivation of a parallel string matching algorithm. Inf. Process. Lett. **85**(5), 255–260 (2003). doi:10.1016/S0020-0190(02)00416-7
32. OpenMPI Web Site: http://www.open-mpi.org. Retrieved Aug 2012
33. Ostermann, S., Iosup, A., Yigitbasi, N., Prodan, R., Fahringer, T., Epema, D.: A performance analysis of EC2 cloud computing services for scientific computing. In: Cloud Computing, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 34, chap. 9, pp. 115–131. Springer, Berlin, Heidelberg (2010)
34. Perl Compatible Regular Expression Library Web Site: http://www.pcre.org. Retrieved Aug 2012
35. PROSITE Web Site: http://prosite.expasy.org. Retrieved Aug 2012
36. Ravikumar, B.: Parallel algorithms for finite automata problems. In: IPPS/SPDP Workshops, vol. 1388. Springer (1998)
37. Raymond, D., Wood, D.: Grail: a C++ library for automata and expressions. J. Symb. Comput. **17**, 17–341 (1995)
38. Roesch, M.: Snort: lightweight intrusion detection for networks. In: Proceedings of the 13th USENIX Conference on System Administration, LISA '99, pp. 229–238. USENIX Association (1999)
39. ScanProsite Web Site: http://prosite.expasy.org/scanprosite. Retrieved Aug 2012
40. Scarpazza, D.P., Villa, O., Petrini, F.: Peak-performance DFA-based string matching on the Cell processor. In: 21th International Parallel and Distributed Processing, Symposium, pp. 1–8 (2007)
41. Schad, J., Dittrich, J., Quiané-Ruiz, J.A.: Runtime measurements in the cloud: observing, analyzing, and reducing variance. Proc. VLDB Endow. **3**(1–2), 460–471 (2010)
42. Sigrist, C., Cerutti, L., De Castro, E., Langendijk-Genevaux, P., Bulliard, V., Bairoch, A., Hulo, N.: PROSITE, a protein domain database for functional characterization and annotation. Nucleic Acids Res. **38**(suppl 1), D161 (2010)
43. Sommer, R., Paxson, V.: Enhancing byte-level network intrusion detection signatures with context. In: Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS '03, pp. 262–271. ACM (2003). doi:10.1145/948109.948145
44. StarCluster cluster computing toolkit: http://web.mit.edu/star/cluster. Retrieved July 2012, version 0.93.3
45. Tewari, A., Srivastava, U., Gupta, P.: A parallel DFA minimization algorithm. In: Proceedings of the 9th International Conference on High Performance Computing, HiPC '02, pp. 34–40. Springer (2002)
46. Wang, G., Ng, T.S.E.: The impact of virtualization on network performance of Amazon EC2 data center. In: Proceedings of the 29th Conference on Information Communications, INFOCOM'10, pp. 1163–1171. IEEE Press (2010)
47. Wang, X., He, K., Liu, B.: Parallel architecture for high throughput DFA-based deep packet inspection. In: 2010 IEEE International Conference on Communications, pp. 1–5 (2010)