Analyzing large-scale DNA Sequences on Multi-core Architectures

(CSE-2015, ©IEEE)

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

***Abstract*—Rapid analysis of DNA sequences is important in preventing the evolution of different viruses and bacteria during an early phase, early diagnosis of genetic predispositions to certain diseases (cancer, cardiovascular diseases), and in DNA forensics. However, real-world DNA sequences may comprise several Gigabytes and the process of DNA analysis demands adequate computational resources to be completed within a reasonable time. In this paper we present a scalable approach for parallel DNA analysis that is based on Finite Automata, and which is suitable for analyzing very large DNA segments. We evaluate our approach for real-world DNA segments of mouse (2.7GB), cat (2.4GB), dog (2.4GB), chicken (1GB), human (3.2GB) and turkey (0.2GB). Experimental results on a dual-socket shared-memory system with 24 physical cores show speedups of up to 17.6×. Our approach is up to $3\times$ faster than a pattern-based parallel approach that uses the RE2 library.**

***Index Terms*—parallel DNA analysis, multi-core architectures, finite automata**

I. Introduction

The need for high performance computational biology has emerged as a result of fast growth in biological information, the complexity of interactions that underlie many processes in biology, as well as the diversity and the interconnectedness of organisms at the molecular level [5]. These biological information are accumulated via different techniques, however they require adequate analysis and processing to extract useful information that make the results evident.

According to Benson et al. [9] the number of Deoxyribonucleic Acid (DNA) sequences and nucleotide bases in these sequences is growing exponentially, doubling every 18 months. As these data are collected, motif search and DNA sequencing are just some examples among many for analytics of Next Gen Sequencing Analysis.

A DNA sequence contains specific genetic instructions, which make the living organisms function properly. In a DNA strand there are four bases of nucleotides: A-adenine, C-cytosine, G-guanine and T-thymine. DNA analysis is important for discovery of differences and similarities of organisms and exploration of the evolutionary relationship between them. This process often requires comparisons of the corresponding DNA sequences, for example, checking whether one sequence is a subsequence of another, or comparing the occurrences of specific k-mers in the corresponding DNA sequences. In computational biology k-mers refer to all the possible sub-strings (sub-sequences) of length k of a DNA sequence. They have an important role during sequence assembly and can be used in sequence alignment as well.

Analyzing DNA sequences within a reasonable time is important for domain scientists to study various phenomenons, such as the evolution of viruses and bacteria during an early phase [20], or diagnosis of genetic predispositions to certain diseases.

Modern parallel computing systems promise to provide the capabilities to cope with the DNA analysis processing requirements. Existing approaches use both hardware and software to accelerate regular expression matching. The hardware based approaches (such as [7], [27]) are faster, but less flexible and more expensive, whereas software based acceleration techniques are flexible in terms of updating or adding new patterns [30]. Recently different software based DNA analysis techniques designed for multi-core systems have been proposed [6], [11], [14], [16], [19], [23].

In this paper, we will first explore and discuss the parallelization opportunities of DNA analysis, and thereafter we introduce a parallel algorithm for DNA analysis that is based on Finite Automata. We use a domain decomposition approach for parallelization; in our approach the DNA sequence is split into several chunks, and each chunk is assigned to a thread to perform pattern matching. Our algorithm is optimized to do efficient speculations of the possible initial states for each chunk. Only one regular expression matching (REM) for a chunk is required to be completely performed; the remaining REMs stop when the converging point is reached. A converging point is a state where two or more REM starting from different states meet after the same number of symbols is read. Furthermore, we use a memory efficient data structure that saves the necessary information to count and highlights the k-mers. Experiments with real-world DNA segments (for human and various animals) on a dual socket shared-memory system with 48 threads show significant speedups compared to the sequential version (up to 17.6x). The implementation of our algorithm is up to 3x faster than a pattern-based algorithm implemented using the RE2 library [3]. Major contributions of this paper include:

- a parallel algorithm for DNA analysis that is based on Finite Automata;
- empirical evaluation of our algorithm with real-world DNA segments of mouse (2.7GB), cat (2.4GB), dog

(2.4GB), chicken (1GB), human (3.2GB) and turkey (0.2GB);
- a comparison of our algorithm with a pattern-based algorithm implementation that uses RE2 library.

The rest of the paper is organized as follows. Section II provides background information on pattern matching, whereas Section III presents our algorithm for counting and extracting k-mers in a DNA sequence. Section IV presents the experimental setup and discusses the experimental results. The work described in this paper is compared and contrasted to the related work in Section V. Section VI provides a summary of our work.

II. Regular Expression Matching (REM) with Finite Automata (FA)

Regular expression matching verifies whether a pattern is present in a string. REM is commonly used for determining the locations of a pattern within a sequence of tokens, in search and replace functions, or to highlight important information out of a huge data set. In the context of computational biology, pattern matching is used for analyzing and processing biological information in order to extract the useful parts of the data and make them evident. The formal definition of the REM is as follows: the input text is an array $T[1..n]$ where n is the length of the input, and pattern $P[1..m]$ where the length of the pattern $m \leq n$. The alphabet $\sum$ defines the possible characters of the input string.

A Finite Automaton (FA) is a machine for processing information by scanning the input text T in order to find the occurrences of the pattern P. A formal definition of the FA is as follows: FA is a quintuple of $(Q, \sum, \delta, q_0, F)$, where Q is the finite set of states, $\sum$ is the finite alphabet, δ is the transition function $Q \times \sum \longrightarrow Q$, q_0 is the start state and F is the distinguished set of final states.

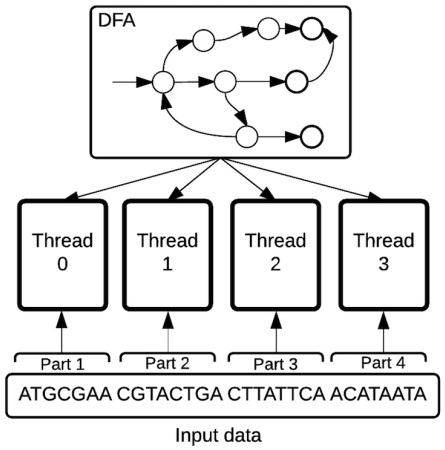
A well known algorithm for multiple pattern matching is the Aho-Corasick algorithm. It is able to match any occurrences (including the overlapped ones) of multiple patterns linearly to the size of the input string. It examines each character of the input string only once. It builds an automaton by creating states and transitions corresponding to these states. It adds failure transitions when there is no regular transition leaving from the current state on a particular character, which makes it possible to match multiple and overlapping occurrences of the patterns. Furthermore, this algorithm is capable of delivering input-independent performance if implemented efficiently in parallel systems, which is a reason why we use this algorithm as basis of our work.

III. Design and Implementation of a DNA Analysis algorithm

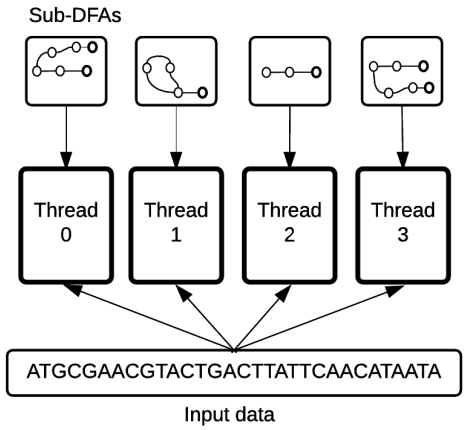
In this section we first provide the details about the outline of our algorithm. Thereafter we discuss the most important implementation aspects to achieve a scalable algorithm for counting and extracting specific k-mers from a large DNA sequence.

A. Our algorithm for counting and extracting the location of k-mers (k-mers CoEx)

Figure 1 depicts two possible ways for parallel execution of regular expression matching for bio-computing applications: (a) *input-based approach* that splits the input string into smaller chunks and processes them in separate threads and (b) *pattern-based approach* that splits the patterns in sub-patterns, creating separate state machines for each of them and processing the same input string with each different machine [16].



(a) Input-based approach



(b) Pattern-based approach

Fig. 1. Load balancing using Input and Pattern partitioning approach.

Our algorithm uses the input-based approach. The challenge of this approach is determining the initial state for each chunk. Finding the correct starting state for each chunk is important for finding the occurrences of the patterns that appear in the crossing border. Other researchers use different ways of finding the initial states, for instance Luchaup et al. [18] use speculation to find the initial state based on the most visited states, Devi and Rajagopalan [12] use an index based technique, Chacon et al. [11] use Suffix-Arrays, Villa et al. [31], uses the pattern length overlapping approach.

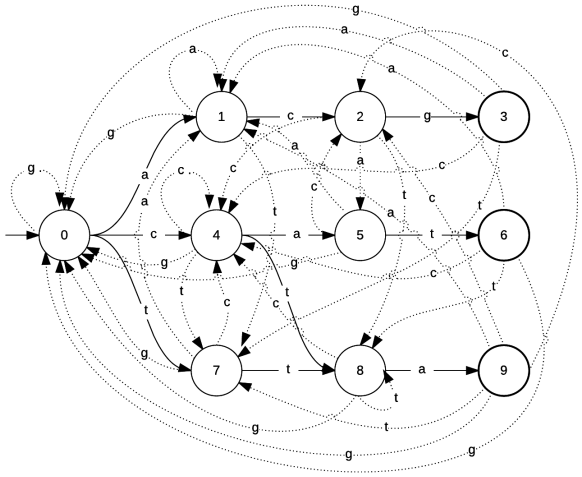


Fig. 2. K-mers CoEx finite state automaton for matching the patterns: "acg", "cat", "cta" and "tta"

Our way of determining the possible initial states is as follows: (1) find the set of source states (L) for the first element of the sub-input mapped to the running thread (T_n); (2) find the set of destination states (S) for the last character of the sub-input mapped to the previous thread (T_{n-1}); (3) find the intersection of S and L ($S \cap L$), which is the set of possible initial states [21]. The first thread (T_0) always starts from the initial state q_0. Each thread is responsible for finding the set of possible initial states, and for each state of this set a regular expression matching is performed. When all threads have finished their job, the results are joined by a binary reduction, which connects the last visited state of T_n to the first visited state of T_{n+1}.

This method provides very good results for sparse transition tables and good performance for dense matrices for DFAs with relatively small number of states. However, in a DFA with large number of states this method seems to be less efficient. This happens because one thread may be responsible to perform multiple REM for the same input, due to multiple possible initial states. To reduce the operations required for each thread to perform the REM starting from different states, further optimizations are needed.

While investigating the REM using the modified Aho-Corasick DFA representation, we noticed that the result converges after several symbols are read (in our experiments, 10 is the max number of steps required to find the converging point). A converging point is a state where two or more REM starting from different states meet after the same number of symbols are examined. This insight allows us to significantly minimize the execution cost required to perform the REM starting from each possible initial state. The details about the process of the convergence are given in the next Section.

B. Implementation Aspects

In this section we will explain the implementation details of our algorithm, including the process of building the DFA, splitting the input among the available threads, finding the set of possible initial states for each chunk assigned to a thread, running the REM for the first state in this set and the process of finding the *converging point*. A reference of each process to the corresponding lines of codes in Algorithm 1 will be provided.

Furthermore, we define q_i as the state of the automaton where i is the state id, E_i as a sub-pattern of the selected patterns, R_i as the process of performing an REM starting from the item at index i of the set of possible initial states. The values used in the *switch-case* (Line 25 - 30) (ex. 122, 127, 128...) determine that a specific sub-pattern (E_i) has been matched.

1) Building the Deterministic Finite Automaton: The AC algorithm with failure transitions has a drawback due to the non-deterministic transitions for a single input character. Figure 2 illustrates our solution to eliminate the failure transitions by adding the right transition (indicated by dashed lines) for each state. Having a valid transition for each possible character to another state in the automaton, guarantees that for each symbol the same amount of operations will be performed. The example automaton shown on Figure 2 is able to match the following patterns: "acg", "cat", "cta" and "tta". For example, if we read string "ac" we reach state q_2, and when "a", "c" or "t" is read we know exactly that state q_6, q_5 or q_{10} is next, respectively.

2) Splitting the input and finding the possible starting states (PSS): The process of splitting the input among the available threads is depicted in Table I.a. This is a straight forward step, where the input length is divided by the number of available threads (Line 5-6). The chunks are assigned to the threads consecutively based on the thread IDs.

The pseudo-code of the process of finding the PSS (See Section III-A) is shown on Algorithm 1 Line 10. When the PSS are determined the thread performs an REM starting from each item of PSS (Line 14). Table I.b,c depicts the REM process starting from each item of PSS; each table corresponds to a thread. For example, the first thread (see Table I.b) initiates the REM starting from the following PSS: q_3, q_{10}, q_{13}, and q_{139}.

For every R_i (REM starting from the PSS at index i) a CR structure is created and stored in the $results$. The CR structure stores the *initial state*, *last state* and the total number of occurrences for each of the sub-patterns (Line 61 - 65).

3) Determining the Converging Point: We investigated that while performing an REM of the same input string starting from different states, the REM converges after a certain number of steps. An example of how the convergence happens is depicted in Table I.b,c. For example in Table I.c, the matching of *CCCTATACGA...* (see Table I.a) starting from q_3 leads to the following transitions: $q_2 \rightarrow q_2 \rightarrow q_2 \rightarrow q_{10} \rightarrow q_{11} \rightarrow q_{130} \rightarrow q_{40} \rightarrow q_{60} \rightarrow q_{15} \rightarrow q_1$.... Matching the same input starting from q_{10} leads to the following transitions: $q_{19} \rightarrow q_2$; no further transitions are needed, because the state at position two for both REMs (R_0 and R_1) is q_2, which indicates the point of convergence.

Algorithm 1 k-mers CoEx

Input: transition table dfa; set of final states F; input string I
Output: list of $CR\ results$

1: **procedure** KCOEX(dfa, F, I)
2: $steps = 10$ ▷ The max number of steps required to converge
3: $result = list < list < CR >>$ ▷ Stores the final states for each thread
4: **for** $T_0 ... T_n$ **do** in parallel ▷ T - Thread, n - total number of threads
5: $start\_position = t\_i * (I.length/n)$ ▷ $t\_i = thread\_id$
6: $SI = substring(start\_position, I.length/n)$ ▷ SI - sub input
7: **if** t_i ! $= 0$ **then**
8: $PSS = get\_possible\_starting\_states(I[start\_position], I[start\_position - 1])$
9: **else**
10: $PSS = [0]$
11: **end if**
12: $fr\_list = list < FR >$
13: $psi\_i = 0$
14: **for** $int\ cs\ in\ PSS$ **do**
15: $CR\ cr$ ▷ stores the init state, last state, and total number of occurrences for each subexpression
16: $char\_i = 0$
17: **for** $char\ c\ in\ SI$ **do**
18: **if** $char\_i == 0$ **then**
19: $cr.init\_state = cs$
20: **else if** $char\_i == SI.length - 1$ **then**
21: $cr.last\_state = dfa[cs][c]$
22: **end if**
23: $cs = dfa[cs][c]$
24: **if** $cs\ in\ F$ **then**
25: **switch** cs **do**
26: **case** 117
27: $cr.final\_states[0] ++$ ▷ $agggtaaa \mid tttaccct$ is found
28: **case** 122 or 128
29: $cr.final\_states[1] ++$ ▷ $(c|g|t)gggtaaa \mid tttaccc(a|c|g)$ is found
30: ...
31: **end if**
32: **if** $psi\_i == 0\ and\ char\_i \leq steps$ **then**
33: $FR\ fr$
34: $fr.current\_state = cs$
35: $fr.final\_states[0] = cr.final\_states[0]$
36: ...
37: $fr.final\_states[8] = cr.final\_states[8]$
38: $fr\_list.add(fr)$
39: **else if** $psi\_i > 0\ and\ fr\_list[char\_i].current\_state == cs$ **then** ▷ check for convergence
40: $cr.final\_states[0] += results[t\_i][0].final\_states[0] - fr\_list[char\_i].final\_states[0]$
41: ...
42: **break**
43: **end if**
44: **end for**
45: $results[t\_i].add(cr)$
46: **end for**
47: **end for**
48: **end procedure**

Input: transition table dfa; the first character of the input mapped to T_n (current thread) $first\_char$; the last character of the input mapped to T_{n-1} (previous thread) $last\_char$
Output: list of states

49: **procedure** GET_POSSIBLE_STARTING_STATES($dfa, first\_char, last\_char$)
50: $S = L = list < q >$ ▷ q - state of the DFA
51: **for** $q_0 ... q_n$ **do**
52: **if** $dfa[q_i][first\_char] \in Q$ **then** ▷ Q-list of states
53: $S_i = q_i$
54: **end if**
55: **if** $dfa[q_i][last\_char] \in Q$ **then**
56: $L_i = dfa[q_i][last\_char]$
57: **end if**
58: **end for**
59: **return** $S \cap L$
60: **end procedure**

61: **struct** CR{
62: **int** $init\_state$
63: **int** $last\_state$
64: **int** $final\_states$[9] ▷ Stores the number of occurrences for each sub-expression (E1...E9)
65: }

66: **struct** FR{
67: **int** $current\_state$
68: **int** $final\_states$[9]
69: }

TABLE I
THE PROCESS OF SPLITTING THE INPUT, PERFORMING THE REM STARTING FROM EACH POSSIBLE INITIAL STATE, AND THE PROCESS OF CONVERGENCE

a) Splitting the input string into chunks

		0	1	2	3	4	5	6	7	8	9		
T_1	→	G	T	G	A	G	C	C	G	A	G	…	chunk 1
T_2	→	C	C	C	T	A	T	A	C	G	A	…	chunk 2

b) REM for chunk 1

Initial state												
1	→	7	21	31	1	7	19	2	9	1	7	…
6	→	16	21	…								
18	→	32	48	13	1	…						
43	→	63	21	…								

c) REM for chunk 2

Initial state												
3	→	2	2	2	10	11	130	40	60	15	1	…
10	→	19	2	…								
44	→	67	90	112	129	1	8	11	5	15	…	
63	→	84	105	2	…							

TABLE II
A TABULAR REPRESENTATION OF THE fr_list (LINE 12)

Step	*CS*	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9
1	67	0	0	0	0	0	0	0	0	0
					...					
4	129	0	0	0	0	0	1	0	0	0

In order to properly count the number of occurrences of k-mers when a converging point is met, we need to store the number of occurrences of k-mers for the first n steps while performing R_0. For each of the first n examined characters a FR structure is created and stored in the fr_list (Line 32-38). The FR structure is shown on Algorithm 1 Line 66 - 69, which stores the *current state* of the automaton, and the current number of occurrences for each of the sub-patterns. An example of this process is depicted in Table II, where each row represents a FR structure. In this example, we assume that q_{44} is the first state in the PSS, which means that the R_0 starts from q_{44}. After four characters ($CCCT$) are examined, a final

TABLE III
THE TABULAR REPRESENTATION OF THE $results$ (LINE 3)

T	Q_0	Q_n	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9
1	1	130	0	0	0	1	0	0	0	0	0
	96	130	0	0	0	1	0	0	0	0	0
2	3	130	0	0	1	0	1	0	2	0	1
	44	130	0	0	1	0	1	1	2	0	1

T - thread index
Q_0 - start state, Q_n - end state
$E_1 - E_9$ - sub-expressions
CS - current state

state (q_{129} - representing E_6, see Table I.e row 3) is reached, therefore the number of current occurrences of k-mers for $E6$ becomes 1 (see Table II).

The REM starting from the remaining states of PSS will be performed until the converging point is reached. For instance (see Table I.b), R_1, R_2 *and* R_3 need only 2, 4, 2 characters to be examined, respectively. When the converging point is reached, the total number of final states ($CR.final\_states$) is calculated by adding the total number of states found for R_0 ($results[t\_i][0].final\_states$) to the R_i and subtracting the final states ($fr\_list[char\_i].final\_states$) found of R_0 at the converging point (Line 41).

IV. Experimental Evaluation

In this section we describe the experimentation environment used for the evaluation of our proposed algorithm and we discuss the obtained performance results.

A. Experimentation Environment

We have performed experiments on a shared-memory system with two 12-core Intel Xeon processors of the type E5-2695 v2 and 16GB Memory. In total the system has 24 physical cores and each physical core supports two threads (also known as logical cores). We have implemented our algorithm using C++11 programming language and OpenMP. For compilation we used the Intel Compiler icc 15.0.0. In order to address the variability in performance measurements we have repeated each experiment 20 times.

For our experimental evaluation we have selected data-sets of genomes from the GenBank National Center for Biotechnology Information sequence database [2]: mouse (2.7GB), cat (2.4GB), dog (2.4GB), chicken (1GB), human (3.2GB) and turkey (0.2GB). The information about the data-sets is provided in Table IV.

TABLE IV
DNA data-sets

	Genome Reference	*Size (MB)*
Mouse	GRCm38.p2	2830
Cat	Felis_catus-6.2	2490
Dog	CanFam3.1	2440
Chicken	Gallus_gullus-4.0	1060
Human	GRCh38	3250
Turkey	Meleagris_gallopavo	193

In our experiments we used the same patterns as in the *regex-dna* benchmark, listed in Table V [1], which are used to extract and match DNA 8-mers and substitute nucleotides according to standards of International Union of Biochemistry (IUB). In Section IV-B we compare the performance of *regex-dna* benchmark with our k-mers CoEx algorithm.

The DFA for the given regular expression (Table V) was generated using our PaREM tool [21]. Figure 3 depicts the DFA of 137 states, which is able to find the occurrences (including the overlapping ones) of the selected patterns. For simplicity the failure links are omitted from the DFA graph.

TABLE V
Patterns of the *regex-dna* benchmark. The symbol "|" determines the "OR" regex operator

$E1$	$agggtaaa$	|	$tttaccct$
$E2$	$(c\|g\|t)gggtaaa$	|	$tttaccc(a\|c\|g)$
$E3$	$a(a\|c\|t)ggtaaa$	|	$tttacc(a\|g\|t)t$
$E4$	$ag(a\|c\|t)gtaaa$	|	$tttac(a\|g\|t)ct$
$E5$	$agg(a\|c\|t)taaa$	|	$ttta(a\|g\|t)cct$
$E6$	$aggg(a\|c\|g)aaa$	|	$ttt(c\|g\|t)ccct$
$E7$	$agggt(c\|g\|t)aa$	|	$tt(a\|c\|g)accct$
$E8$	$agggta(c\|g\|t)a$	|	$t(a\|c\|g)taccct$
$E9$	$agggtaa(c\|g\|t)$	|	$(a\|c\|g)ttaccct$

B. Results

We first present the performance results of our k-mers CoEx algorithm for various problem and machine sizes, and thereafter we compare our algorithm with a pattern-based algorithm implementation that uses RE2 library (known as *regex-dna* benchmark). Figure 4 depicts the execution time in logarithmic scale for each of our selected data-sets and for various numbers of threads {1,2,6,12,24,48}. We observe a good scalability of our algorithm as we increase the number of threads or the input size. For example, the analysis of the human's DNA sequence using one thread takes 27 seconds, and by increasing the number of threads to 2, 6, 12, 24 and 48 the execution time reduces to 14.3s, 5.3s, 3s, 2s and 1.5s respectively.

Figure 5 depicts the obtained speedup of our algorithm compared to a sequential version of the Aho-Corasick algorithm for DNA analysis. We may observe that the k-mers CoEx algorithm scales gracefully with respect to the size of data-sets and the number of threads. The maximal speedup of 17.65× is achieved for the largest data-set (that is the human DNA segment) using 48 threads.

Figure 6 compares the performance of our k-mers CoEx algorithm with the *regex-dna benchmark* [1], which is implemented in C++ using the RE2 library [3] and OpenMP. The RE2 implementation is based on splitting the pattern in smaller patterns, and matching the input string in parallel for each sub-pattern. Since the *regex-dna benchmark* does not support larger data-sets, we have compared the two algorithms for the two smallest data-sets: chicken (1060MB) and turkey (193MB). Our k-mers CoEx algorithm outperforms the *regex-dna benchmark* for both data-sets. We may observe that the k-mers CoEx algorithm running on one thread takes the same amount of time as the *regex-dna benchmark* running on the total amount of threads. This happens because one thread has to perform at least one sequential REM for a specific sub-pattern. In this class of algorithms where the execution time is mainly dependent on the length of the input, balancing the work among the available threads should be done by splitting the input string, instead of the pattern length. One could benefit from partitioning a long pattern into smaller one, in cases when the input string is either relatively short or can not be split (Real-time Network Intrusion Detection).

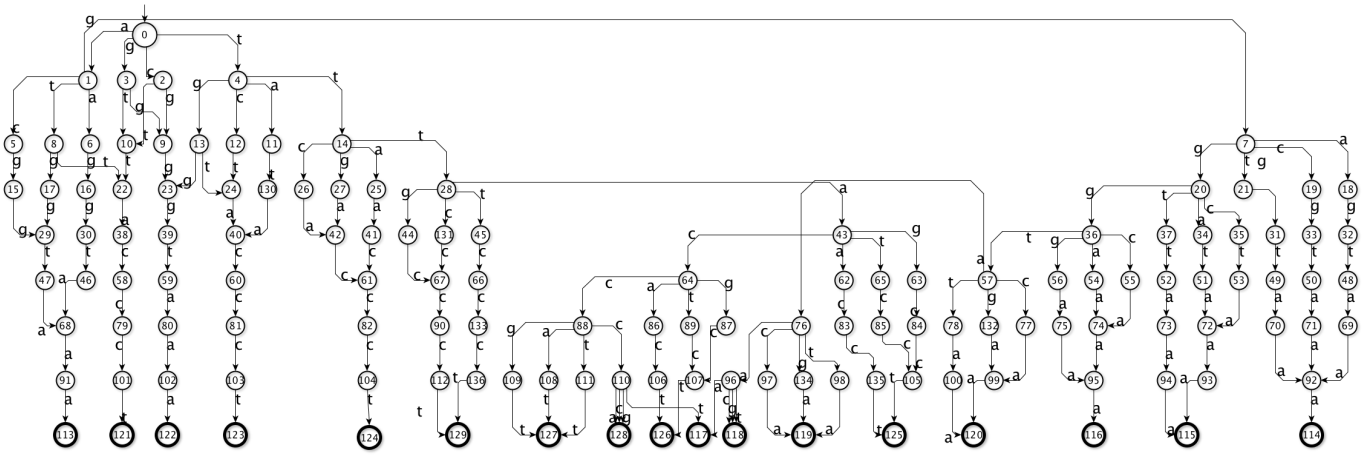


Fig. 3. The DFA automaton that matches (counts and extracts the location of k-mers) 8-mers from a given DNA sequence. The corresponding regular expression is shown on Table V. For simplicity the failure links are omitted from the figure.

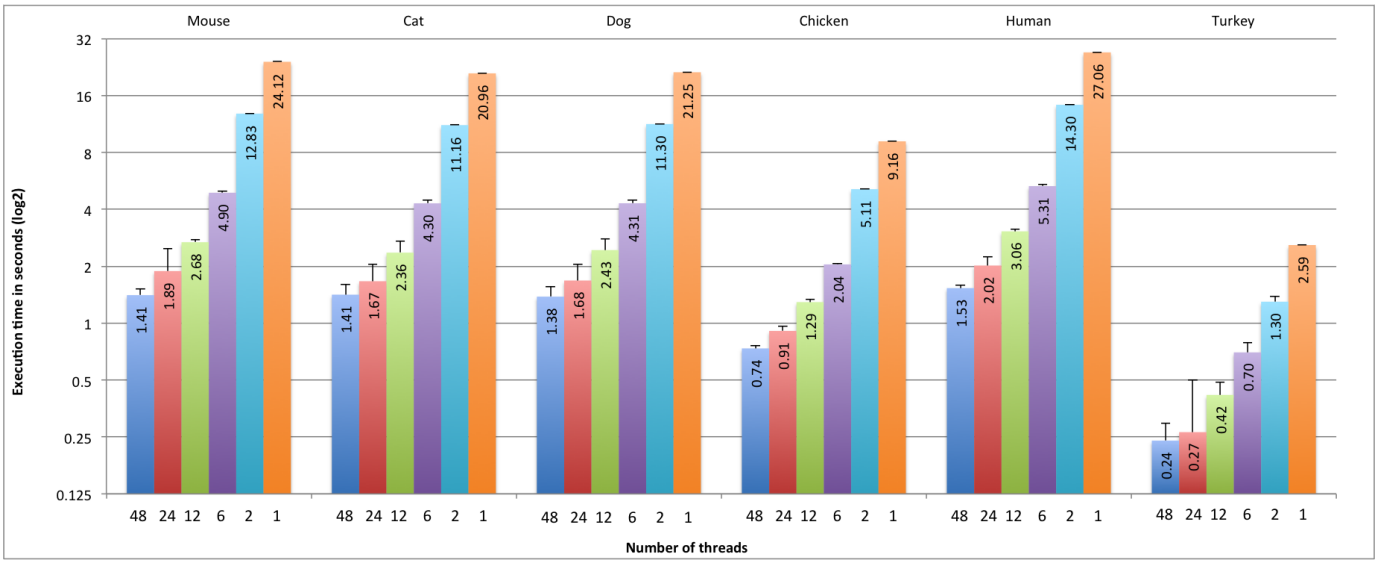


Fig. 4. Performance results of our k-mers CoEx algorithm for various numbers of threads and data-sets. As input are used six DNA sequences of various lengths: mouse (2.7GB), cat (2.4GB), dog (2.4GB), chicken (1GB), human (3.2GB), and turkey (0.2GB). The experiments are performed by varying the number of threads (2, 6, 12, 24, and 48). The performance measurements for each experiment have been repeated 20 times.

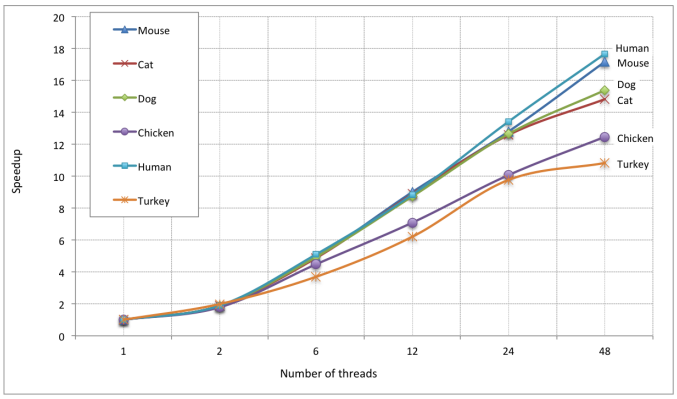


Fig. 5. Speedup of our k-mers CoEx algorithm implementation.

V. Related Work

In this section we discuss the state-of-the-art in pattern matching and DNA sequence analysis techniques for multi-core architectures.

Existing approaches use both hardware and software to accelerate the process of regular expression matching. In comparison to hardware based state machines, which are faster, less flexible and more expensive, software based acceleration techniques are flexible in terms of updating or adding new patterns [30].

Herath et al. presented in [16] an implementation of the Aho-Corasick string matching algorithm using POSIX threads, which is based on the pattern partitioning approach. A replication of the Herath's study with the intention to improve the

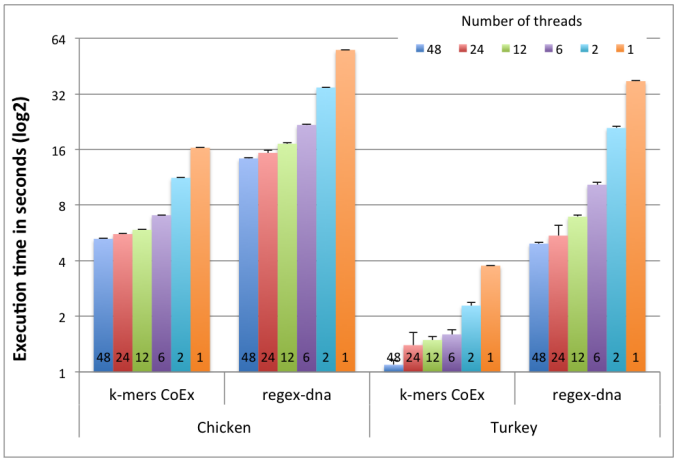


Fig. 6. Performance comparison between k-mers CoEx and the *regex-dna benchmark* (RE2)

software implementation of the Aho-Corasick algorithm was conducted by Arudchutha et al. [6].

Marçais and Kingsford [19] present the Jellyfish tool, which is based on the lock-free hash table that is optimized for counting k-mers of length up to 31 bases. Rizk et al. [28] present a similar approach to Jellyfish [19], so called DSK, which is designed for small-memory servers. The k-mers are counted by traversing the hash tables. Using hash tables for the internal representation resulted to be memory inefficient [14]. As described by Drews et al. [14] a sequence corresponding to a human chromosome with 24-230MB of input data would require gigabytes of memory to store the k-mers information.

Drews et al. [14] achieved significant speedup by partitioning the input string among the threads in such a way that each thread processes only sequences starting with a specified prefix used to divide the radix tree among the threads. They achieved up to 6.9× speedup on a shared memory system with 8 cores.

The n-step FM-index approach presented by Chacón et al. [11] achieved speedups from 1.4× to 2.4× with respect to their original FM-index search algorithm.

An approach based on the Aho-Corasick string matching algorithm designed for the Cray XMT architecture is proposed by Villa et al. [31]. They split the input among the available threads, and overlap the input by the pattern length. Their approach is applicable for multiple patterns as long as they are of the same length, otherwise, the occurrences of the shortest patterns occurring on the crossing border may be counted by both threads.

A method for searching arbitrary regular expressions using speculation is proposed by Luchaup et al. [18]. The drawback is that if an REM performed by a thread does not converge on its sub-input, then the next thread has to start from a new state that breaks the serialization and limits the scalability.

Our k-mers CoEx algorithm is tailored for large-scale DNA analysis. In our approach, the DNA segment is split into several chunks, and efficient speculations of the possible initial states for each chunk are performed. Furthermore, our algorithm optimizes the REM using a converging point.

VI. Summary

We have described a parallel algorithm based on Finite Automata for counting and extracting k-mers in a DNA segment. In a series of experiments with real world data-sets we have observed that the algorithm scales well with respect to various problem and machine sizes. We achieved the maximal speedup of 17.65× for the largest data-set (that is the human DNA segment) using 48 threads on a dual-socket shared-memory system with 24 physical cores. In comparison to the *regex-dna benchmark* our algorithm was up to three times faster.

In this paper we have studied the performance of our approach for DNA sequence analysis on a shared-memory system with two 12-core Intel Xeon processors. It may be useful to compare the performance that we achieved using all available cores of the host Intel Xeon processors with the performance achieved when all available cores of the Intel Xeon Phi coprocessor are used [22]. Furthermore, software technologies, such as [13], enable the use of all cores of homogeneous processors of the host and all available cores of the coprocessor.

Future work could address generalization of our approach for DNA sequence analysis for various types of accelerated systems using techniques that ensure performance portability [8], [17], [24], [29]. The use of modeling and simulation techniques [10], [15], [25], [26] could help to reason about the performance on extreme-scale computing architectures [4].

References

[1] The computer language benchmarks game. http://benchmarksgame.alioth.debian.org/. Accessed: Jan. 2015.

[2] National center for biotechnology information u.s. national library of medicine. http://www.ncbi.nlm.nih.gov/genbank. Accessed: Jan. 2015.

[3] Re2: An efficient, principled regular expression library. https://code.google.com/p/re2/. Accessed: Jan. 2015.

[4] E. Abraham, C. Bekas, I. Brandic, S. Genaim, E. B. Johnsen, I. Kondov, S. Pllana, and A. Streit. Preparing hpc applications for exascale: Challenges and recommendations. In *2015 International Conference on Network-Based Information Systems (NBiS)*. IEEE, 2015.

[5] S. Aluru, N. M. Amato, and D. A. Bader. Editorial: Special section on high-performance computational biology. *IEEE Transactions on Parallel and Distributed Systems*, 17(8):737–739, 2006.

[6] S. Arudchutha, T. Nishanthy, and R. G. Ragel. String matching with multicore cpus: Performing better with the aho-corasick algorithm. *arXiv preprint arXiv:1403.1305*, 2014.

[7] Y. Benenson, T. Paz-Elizur, R. Adar, E. Keinan, Z. Livneh, and E. Shapiro. Programmable and autonomous computing machine made of biomolecules. *Nature*, 414(6862):430–434, 11 2001.

[8] S. Benkner, S. Pllana, J. Traff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, and V. Osipov. PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems. *Micro, IEEE*, 31(5):28–41, Sept 2011.

[9] D. A. Benson, M. Cavanaugh, K. Clark, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and E. W. Sayers. Genbank. *Nucleic acids research*, 41(D1):D36–D42, 2013.

[10] I. Brandic, S. Pllana, and S. Benkner. An approach for the high-level specification of qos-aware grid workflows considering location affinity. *Scientific Programming*, 14(3–4):231–250, 2006.

[11] A. Chacn, J. C. Moure, A. Espinosa, and P. Hernndez. n-step fm-index for faster pattern matching. In V. N. Alexandrov, M. Lees, V. V. Krzhizhanovskaya, J. Dongarra, and P. M. A. Sloot, editors, *ICCS*, volume 18 of *Procedia Computer Science*, pages 70–79. Elsevier, 2013.

[12] S. N. Devi and S. P. Rajagopalan. Article: An index based pattern matching using multithreading. *International Journal of Computer Applications*, 50(6):13–17, July 2012.

[13] J. Dokulil, E. Bajrovic, S. Benkner, S. Pllana, M. Sandrieser, and B. Bachmayer. High-level support for hybrid parallel execution of c++ applications targeting intel xeon phi coprocessors. *Procedia Computer Science*, 18(0):2508 – 2511, 2013. 2013 International Conference on Computational Science.

[14] F. Drews, J. Lichtenberg, and L. R. Welch. Scalable parallel word search in multicore/multiprocessor systems. *The Journal of Supercomputing*, 51(1):58–75, 2010.

[15] T. Fahringer, S. Pllana, and J. Testori. Teuta: Tool support for performance modeling of distributed and parallel applications. In *Computational Science - ICCS 2004*, volume 3038 of *Lecture Notes in Computer Science*, pages 456–463. Springer Berlin Heidelberg, 2004.

[16] D. Herath, C. Lakmali, and R. Ragel. Accelerating string matching for bio-computing applications on multi-core cpus. In *Industrial and Information Systems (ICIIS), 2012 7th IEEE International Conference on*, pages 1–6, Aug 2012.

[17] C. Kessler, U. Dastgeer, S. Thibault, R. Namyst, A. Richards, U. Dolinsky, S. Benkner, J. Traff, and S. Pllana. Programmability and performance portability aspects of heterogeneous multi-/manycore systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 1403–1408, March 2012.

[18] D. Luchaup, R. Smith, C. Estan, and S. Jha. Speculative parallel pattern matching. *IEEE Transactions on Information Forensics and Security*, 6(2):438–451, 2011.

[19] G. Marais and C. Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

[20] A. Mellmann and et al. Prospective genomic characterization of the german enterohemorrhagic escherichia coli o104:h4 outbreak by rapid next generation sequencing technology. *PLoS ONE*, 6(7):e22751, 07 2011.

[21] S. Memeti and S. Pllana. Parem: A novel approach for parallel regular expression matching. In *Computational Science and Engineering (CSE), 2014 IEEE 17th International Conference on*, pages 690–697, Dec 2014.

[22] S. Memeti and S. Pllana. Accelerating dna sequence analysis using intel xeon phi. In *2015 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*. IEEE, 2015.

[23] K. J. Nordström, M. C. Albani, G. V. James, C. Gutjahr, B. Hartwig, F. Turck, U. Paszkowski, G. Coupland, and K. Schneeberger. Mutation identification by direct comparison of whole-genome sequencing data from mutant and wild-type individuals using k-mers. *Nature biotechnology*, 31(4):325–330, 2013.

[24] S. Pllana, S. Benkner, E. Mehofer, L. Natvig, and F. Xhafa. Towards an intelligent environment for programming multi-core computing systems. In *Euro-Par 2008 Workshops - Parallel Processing*, volume 5415 of *Lecture Notes in Computer Science*, pages 141–151. Springer Berlin Heidelberg, 2009.

[25] S. Pllana, S. Benkner, F. Xhafa, and L. Barolli. Hybrid performance modeling and prediction of large-scale computing systems. In *Complex, Intelligent and Software Intensive Systems, 2008. CISIS 2008. International Conference on*, pages 132–138, March 2008.

[26] S. Pllana, I. Brandic, and S. Benkner. A Survey of the State of the Art in Performance Modeling and Prediction of Parallel and Distributed Computing Systems. *International Journal of Computational Intelligence Research (IJCIR)*, 4(1):17–26, January 2008.

[27] J. Reif and S. Sahu. Autonomous programmable nanorobotic devices using dnazymes. In M. Garzon and H. Yan, editors, *DNA Computing*, volume 4848 of *Lecture Notes in Computer Science*, pages 66–78. Springer Berlin Heidelberg, 2008.

[28] G. Rizk, D. Lavenier, and R. Chikhi. Dsk: k-mer counting with very low memory usage. *Bioinformatics*, 29(5):652–653, 2013.

[29] M. Sandrieser, S. Benkner, and S. Pllana. Using Explicit Platform Descriptions to Support Programming of Heterogeneous Many-Core Systems. *Parallel Computing*, 38(1-2):52–56, January 2012.

[30] B. Soewito and N. Weng. Methodology for evaluating DNA pattern searching algorithms on multiprocessor. In *Proceedings of the 7th IEEE International Conference on Bioinformatics and Bioengineering, BIBE 2007, October 14-17, 2007, Harvard Medical School, Boston, MA, USA*, pages 570–577, 2007.

[31] O. Villa, D. G. Chavarra-Miranda, and K. J. Maschhoff. Input-independent, scalable and fast string matching on the cray xmt. In *IPDPS*. IEEE, 2009.