STUDIA INFORMATICA 2014
Volume 34 Number 4 (118)

Tomasz JASTRZĄB, Zbigniew J. CZECH
Silesian University of Technology, Institute of Informatics

A PARALLEL ALGORITHM FOR THE DECOMPOSITION OF FINITE LANGUAGES

Summary. A finite language is said to be decomposable, if it can be written as a catenation of two non-empty languages. In this paper a parallel algorithm for finding the decomposition of finite languages is proposed. The effectiveness of the algorithm is assessed based on the experimental results provided for selected languages.

Keywords: parallel algorithms, finite languages, finite languages decomposition

RÓWNOLEGŁY ALGORYTM DEKOMPOZYCJI JĘZYKÓW SKOŃCZONYCH

Streszczenie. Język skończony jest dekomponowalny, jeżeli może zostać zapisany jako złożenie dwóch niepustych języków. W niniejszym artykule zaproponowany został równoległy algorytm dekompozycji języków skończonych. Efektywność przedstawionego algorytmu została oceniona na podstawie eksperymentów przeprowadzonych dla wybranych języków.

Słowa kluczowe: algorytmy równoległe, języki skończone, dekompozycja języków skończonych

1. Introduction

A language is said to be finite, if it consists of a finite set of words over a certain alphabet. Since every finite language is regular [9] it may be represented by a finite automaton [11]. Consequently, they may be applied in such areas as lexical analysis in compilers, pattern matching, spell-checking [1, 11], computational biology [7]. Finite languages and their decompositions are also essential to practical applications in the field of grammatical inference [3, 8].

The problem of determining whether a finite language possesses a decomposition is proven to be intractable [4], yet decidable [9]. However, the NP-hardness of this problem has been left as an open issue [4]. Due to lack of the universal solution for finding a decomposition of a language of arbitrary length, the only method left to be used is the exhaustive search of all possible solutions [4]. It was shown in [10] that other approaches may be applied, although they usually fail to provide satisfactory results in terms of correctness. For each finite decomposable language it is possible to obtain a decomposition into non-trivial languages, although this decomposition may not be unique. Furthermore, as it was shown in [4, 6] the decomposition is usually noncommutative, which causes certain difficulties [5]. An indecomposable language is called prime, by analogy to prime numbers in the number theory.

The aim of this paper is to propose a parallel algorithm for the decomposition of finite languages. The algorithm searches for all possible decompositions, not only allowing to state whether a given language is decomposable. It also returns the information about actual decomposition set(s) and factor languages building the input language. Performance of the algorithm is to be compared with existing sequential algorithm based on exhaustive search. In particular, its effectiveness in finding all possible decompositions of a given language is to be assessed for selected languages, which may be considered hard to decompose. In this context a language is considered hard if the space of possible solutions cannot be searched in a reasonable time.

The paper is organized into six sections. In section 2 the basic notions and terms are introduced. Section 3 provides an overview of existing algorithms which constitute the basis for the proposed algorithm, which is later presented in section 4. In section 5 some remarks related to the analyzed languages' structure are presented and the experimental results of decompositions searching are described. Section 6 contains the summary of the paper.

2. Basic terms and notions

An alphabet, denoted by Σ, is understood as a nonempty set of symbols, building words w of a given language L. The length of a word w is denoted by $|w|$. It equals to the number of symbols from the alphabet appearing in the word w. The cardinality of language L, i.e. the number of words in L, is denoted by $|L|$. The empty word is denoted by λ. The length of the empty word is equal 0. A language is called *trivial* or *singleton*, if it consists solely of the empty word λ. For each word w a *prefix* (respectively a *suffix*) is defined as a word $v \in \Sigma^*$ such that $w = vu$, $u \in \Sigma^*$ (resp. $w = uv, u \in \Sigma^*$). A prefix (resp. suffix) is called proper, if $v \neq \lambda$ holds. The symbol Σ^* denotes all the words that may be generated over an alphabet Σ.

Given two words $u, v \in \Sigma^*$ a result of a catenation operation performed on them is defined as a new word $w = uv, w \in \Sigma^*$, produced by first copying word u and then following it by a copy of word v. A catenation of two sets of words $U, V \subset \Sigma^*$ is defined as a set $UV = \{uv \mid u \in U \wedge v \in V\}$. Given word u and set $W \subset \Sigma^*$ one may define *left* (resp. *right*) *quotient* of W as $u^{-1}W = \{w \mid uw \in W\}$ (resp. $Wu^{-1} = \{w \mid wu \in W\}$).

A finite language may be represented in the form of an automaton having the following characteristics:

- It is *minimal*, what means that for all states $p, q \in Q, p \neq q$, the sets of paths, beginning at state p or state q and reaching final states, are not equal [7].
- It is *acyclic*, which follows from the finite nature of the language accepted by the automaton.
- It is *deterministic*, meaning that for all states $p, q \in Q, p \neq q$, different words are spelled out on the paths from the starting state s to given state p or state q. It also implies the existence of only single starting state s [7].
- It is *finite*, what means that it contains a finite set of states.

More formally, a minimal acyclic deterministic finite automaton (MADFA) is defined as a quintuple $(Q, \Sigma, \delta, s, F)$, where:

- Q is the finite set of states of the automaton,
- Σ is an alphabet,
- $\delta : Q \times \Sigma \to Q$ is a transition function, which does not necessarily have to be *total* [11], i.e. it does not have to be defined for all possible elements of the Cartesian product of $Q \times \Sigma$,
- $s \in Q$ is the starting state, and
- $F \subseteq Q$ is the set of final states.

Given a language L, the problem of language decomposition may be formulated as the problem of finding the nontrivial languages L_1 and L_2 such that:

$$L = L_1 L_2 \tag{1}$$

As stated in [2, 10] the problem may be also formulated as the problem of finding a non-empty subset of states $P \subseteq Q$ of the MADFA accepting language L, which satisfies the following condition:

$$L = R_1^P R_2^P \tag{2}$$

where R_1^P, R_2^P are defined as follows:

$$R_1^P = \bigcup_{p \in P} \overleftarrow{p} \tag{3}$$

$$R_2^P = \bigcap_{p \in P} \overrightarrow{p} \tag{4}$$

where $\overleftarrow{p}$ is the left language for the state p, and $\overrightarrow{p}$ is the right language for the state p. The *left language* for state p represents the set of words spelled out on the paths from the starting state s to state p. Respectively the *right language* for state p is the set of words spelled out on the paths from state p to final states. It was proven in [6] that for languages L_i, R_i^P from (1) and (2) it holds $L_i \subseteq R_i^P$, $i = 1,2$.

3. Related works

3.1. Sequential algorithm

In [9] a sequential algorithm for determining the decomposition of a given language based on the exhaustive search with pruning was proposed. The modified version of this algorithm, searching for all available decompositions, is shown in fig. 1. It uses a previously built minimal acyclic deterministic finite automaton, representing the language L being decomposed, by making use of the algorithm presented in [8]. A *significant state* q is a state satisfying one of the following conditions [9]:

- a number of outgoing transitions from state q is at least equal to 2,
- state q is final with at least one outgoing transition.

The variable $inSt$ represents the set of pairs (w, st(w)), where word $w \in L$ and st(w) is the set of significant states for w. Initially, $inSt$ contains all words and by means of respective st(w) it covers altogether all significant states in MADFA. Variable $decSt$ represents the current decomposition set. This set is initially empty and then is successively updated in line 17 during the recursive call of function *decompose*. The pruning of search space is performed in line 16, based on a conclusion following from formula (4). Namely, for the decomposition set $decSt$ and language L, each word w can be written as a catenation of two subwords generated by splitting the input word in the position of state $q \in decSt$. Thus it should be possible to generate the suffix of w in state q by means of at least one state from $decSt$. Consequently all the states in $inSt$ which are not able to produce such a suffix can be removed from the search space. The verification whether a given set of states is the decomposition set is performed in lines 2-7, after the number of additional states falls at or below cut-off level K, which is the parameter of the algorithm.

```
    function decompose(inSt, decSt)
1:     if |st(inSt) – decSt| <= K then                // a cut-off point reached
2:        for each subset D in (st(inSt) – decSt) do  // power set generated
3:           generate P = sum(decSt, D)               // decSt complemented with D
4:           if P is decomposition set then           // formulae (2), (3), (4) used
5:              print P, L1, L2
6:           end if
7:        end for
8:     else
9:        sort inSt according to |st(inSt[i])| in ascending order
10:       if |st(inSt[0])| = 0 then                   // inSt empty
11:          return
12:       else
13:          remove inSt[0] from inSt                  // word and its states removed
14:          for each state q in st(inSt[0]) do
15:             find suffix s of w(inSt[0])            // suffix at state q
             // states not generating s removed
16:             remove redundant states based on s
17:             decompose(inSt, sum(decSt, q))         // decSt extended with q
18:          end for
19:       end if
20:    end if
```

Fig. 1. Sequential algorithm for finite language decomposition
Rys. 1. Sekwencyjny algorytm dekompozycji języka skończonego

3.2. Parallel algorithm

As it was pointed out in [2], a certain amount of time is spent on execution below the cut-off level defined by the value of parameter K (lines 2-7 in fig. 1). Since the generation of the power set of additional states (line 2) and verification whether a current set *P* constitutes the decomposition set (line 4) can be performed independently, this part of the algorithm could be parallelized, as it was done in [2]. The approach taken in [2] involved the master-workers scheme of work distribution with data buffering in the master in order to reduce the amount of communication between master and worker processes. The pseudo-code of this algorithm is shown in fig. 2 and 3 for the master and worker processes respectively. Lines 9-18 in fig. 2 contain the operations presented also in lines 9-19 in fig. 1. Instead of checking the decomposition set in the master process buffering is introduced in lines 2-3. The complete buffer is sent to one of the workers according to round robin communication scheme (lines 4-7). The variable *buffer_size* is adjusted so as to achieve a high communication performance. The worker processes presented in fig. 3 receive potential decomposition sets and perform verification depicted in fig. 1 in lines 3-6.

It was stated in [2] that the applied approach was not satisfactory in terms of speedup. One of the potential drawbacks of the approach was unbalanced loading of worker processes. As the tests have shown the communication between master and workers could also be a source of degradation of performance. Both of these issues were addressed in the improved algorithm proposed in this paper, as described in the next section.

```
    function decomposeM(inSt, decSt)
```

```
if |st(inSt) – decSt| <= K then                // a cut-off point reached
   put pair (st(inSt), decSt) into buffer
   buffered_count = buffered_count + 1         // pairs' counter updated
   if buffered_count = buffer_size then        // a sending point reached
      send buffer to next process in round robin fashion
      buffered_count = 0                       // counter reset
   end if
else
   sort inSt according to |st(inSt[i])| in ascending order
   if |st (inSt[0])| = 0 then                  // inSt empty
      return
   else
      remove inSt[0] from inSt                 // word and its states removed
      for each state q in st(inSt[0]) do
         find suffix s of w(inSt[0])           // suffix at state q
      // states not generating s removed
         remove redundant states basing on s
         decomposeM(inSt, sum(decSt, q))       // decSt extended with q
      end for
      if buffered_count != 0 then              // some pairs not sent yet
         send remaining pairs
      end if
      send finish signal                       // master finished
   end if
end if
```

Fig. 2. Master process used in parallel algorithm
Rys. 2. Proces zarządcy wykorzystywany w algorytmie równoległym

```
function decomposeW()
   while true do
      receive buffer from master          // pairs (st(inSt), decSt) or finish
      if buffer = finish signal then      // finish signal received
         break                            // worker finished
      else
         for each pair Pr in buffer do
         // power set generated for every pair (st(inSt), decSt)
            for each subset D in (Pr[i].inSt – Pr[i].decSt) do
               generate P = sum(Pr[i].decSt, D)  // decSt complemented with D
               if P is decomposition set then    // formulae (2), (3), (4) used
                  print P, L1, L2
               end if
            end for
         end for
      end if
   end while
```

Fig. 3. Worker process used in parallel algorithm
Rys. 3. Proces wykonawcy wykorzystywany w algorytmie równoległym

4. An improved parallel algorithm

Following the idea of parallelization of computation below the cut-off level suggested in [2], an improved algorithm is proposed in this paper aiming at achieving better speedup values. The main assumptions of the algorithm presented in fig. 4 are as follows:

- removal of communication overhead introduced by the master-workers scheme,

- balancing the load of each process by equally distributing the computations regarding the verification steps among processes.

The first aim was achieved by allowing all the processes to execute the code of function *decomposeP* in parallel, so that there is no need for communication between processes. In this scheme each process executes the same code, and consequently it has access to the same data, as if executing in a shared memory model, although the computations are distributed. The second aim was achieved below the cut-off level where each potential decomposition set is produced and checked by a separate process, depending on process number (lines 3-8). This way one takes advantage of the search independence, assuring the correctness of the solution at the same time (by verifying all necessary combinations).

```
    function decomposeP(inSt, decSt)
1:     if |st(inSt) – decSt| <= K then               // a cut-off point reached
2:        for each subset D in (st(inSt) – decSt) do // power set generated
          // sum of decSt and current D verified only in process
                 // with rank matching combination number
3:           if (combination_no++ % proc_cnt) = proc_no then
4:              generate P = sum(decSt, D)           // decSt complemented with D
5:              if P is decomposition set then       // formulae (2), (3), (4) used
6:                 print P, L1, L2
7:              end if
8:           end if
9:        end for
10:    else
11:       sort inSt according to |st(inSt[i])| in ascending order
12:       if |st(inSt[0])| = 0 then                 // inSt empty
13:          return
14:       else
15:          remove inSt[0] from inSt               // word and its states removed
16:          for each state q in st(inSt[0]) do
17:             find suffix s of w(inSt[0])         // suffix at state q
             // states not generating s removed
18:             remove redundant states basing on s
19:             decomposeP(inSt, sum(decSt, q))     // decSt extended with q
20:          end for
21:       end if
22:    end if
```

Fig. 4. Improved parallel algorithm for finite language decomposition
Rys. 4. Ulepszony równoległy algorytm dekompozycji języka skończonego

5. The experiments

5.1. Sample languages

The four decomposable languages were selected for the purpose of algorithms performance assessment. They differed from each other in the number of words ranging from 800 up to 6583 (see tab. 1, column $|L|$). All languages were generated over the alphabet consisting of at most four different symbols i.e. small letters *a*, *b*, *c* and *d* (alphabet size is shown in column $|\Sigma|$). The appropriate minimal acyclic deterministic finite automata were constructed for each language (the minimal number of states are shown in the column labeled |*states*|). Among the states of MADFA, the final and significant states were distinguished, and their counts are presented in columns |*f_states*|, |*s_states*|, respectively.

Table 1

Sample languages characteristics

			MADFA		
Name	$\lvert L\rvert$	$\lvert\Sigma\rvert$	|*states*|	|*s_states*|	|*f_states*|
we-800	800	2	18	17	14
we-5317	5317	4	79	73	37
we-6034	6034	4	74	67	32
we-6583	6583	4	82	74	36

5.2. Experimental results

The algorithms were implemented in C language with the use of Message Passing Interface (MPI). The interpreter run on Intel Xeon Quad Core 2,33 GHz processors, with the nodes interconnected with Infiniband 20 Gb/s network. The operating system was Debian GNU/Linux 4.0.

According to the best knowledge of the authors there are no benchmarks for the problem of finite languages decomposition. The computation times for sequential and parallel algorithms described in sections 3.1 and 4 are given in tab. 2. The column T_{dec} shows the decomposition procedure execution time (without the time of MADFA construction), and T_{belowK} the execution time below the cut-off level. The value of cut-off level parameter (K in fig. 1 and 4) was 10. The number of nodes executing the decomposition algorithm is reported in column *n_count*. The values presented in tab. 2 are the average values out of three measurements performed for each language and node count. All times listed in tab. 2 are given in seconds and were measured using the *clock()* function.

Based on the experimental results presented in tab. 2 one may observe that the performance of the improved algorithm varies depending on the language. The speedup values are highly different among the languages. The best speedups are summarized in tab. 3 in terms of overall algorithm execution times, with node counts for these situations provided in parentheses. Speedup values were computed according to the formula:

$$S(n) = \frac{T(1)}{T(n)} \tag{5}$$

where $S(n)$ denotes the speedup obtained for n nodes, $T(1)$ is the sequential execution time and $T(n)$ is the parallel execution time for n nodes.

Table 2

Execution times for finite languages decomposition

	we-800		we-5317		we-6034		we-6583	
n_count	T_{dec}	T_{belowK}	T_{dec}	T_{belowK}	T_{dec}	T_{belowK}	T_{dec}	T_{belowK}
1	117,25	117,12	921,65	211,40	142,12	135,23	1006,21	157,42
2	60,01	59,89	816,72	106,47	74,75	67,89	887,54	79,27
4	30,76	30,61	766,50	54,15	40,85	34,01	848,03	39,78
6	20,94	20,80	748,78	36,55	30,00	23,07	837,51	26,97
8	16,03	15,90	740,48	27,76	24,10	17,24	828,75	20,18
10	13,09	12,96	734,88	22,59	20,93	14,07	824,66	16,35
12	11,11	10,98	736,56	19,01	18,70	11,82	822,21	13,89
14	9,79	9,65	729,34	16,64	17,05	10,15	820,43	12,11
16	8,72	8,60	726,54	14,55	15,66	8,78	819,00	10,41
18	7,86	7,73	725,37	13,30	14,92	8,04	818,35	9,59
20	7,22	7,10	724,60	12,07	14,21	7,30	817,26	8,71
22	6,67	6,53	723,39	11,18	13,61	6,75	816,51	7,99

Table 3

Best speedup values

we-800	we-5317	we-6034	we-6583
17,87 (22)	1,29 (22)	10,70 (22)	1,24 (22)

The best speedup values shown in tab. 3 were obtained for 22 nodes regardless of the language, but the actual results are quite language-dependent. It follows that in case of languages for which the execution time below cut-off level was close to the total execution time, the speedup values are relatively good. However, for the other languages they are certainly not satisfactory. The value of parameter K affects to a large degree the results obtained by the algorithm. Depending on the size of the part devoted to the decomposition sets verification (which is parallelized) one obtains better or worse performance.

The load distribution in the proposed algorithm is quite balanced, what results in almost linear speedups in terms of execution times below the cut-off level. However, some differences occur, and the potential reason for this may be that with more nodes the distribution of load among processes becomes unequal. It may happen that some of the processes have larger decomposition sets assigned every time. Such conclusions were drawn as the result of detailed analysis of processing times.

6. Final remarks

The paper introduced an improved parallel algorithm for the finite languages decomposition. The main goal was to improve the performance of finding all possible decompositions of a given language. The algorithm is based on the exhaustive search with pruning. The communication between processes in the algorithm is minimized by simultaneously executing sequential searching and selecting solutions to be verified in given process. The experimental results show that the algorithm should still be improved, so as to provide larger speedup values. The potential improvement paths may involve further balancing of load distribution combined with optimal cut-off level choice. It may be crucial to find the relation between the value of parameter K and the distribution of potential decompositions sets among processes. It may be also vital to base the load distribution on the size of processed data instead of assuming every-*n*th decomposition assignment scenario. Another direction of improvement could be the parallelization of the sequential part of the solution space pruning.

Acknowledgements

Calculations were carried out using the computer cluster Ziemowit (http://ziemowit.hpc.polsl.pl) funded by the Silesian BIO-FARMA project No. POIG.02.01.00-00-166/08 in the Computational Biology and Bioinformatics Laboratory of the Biotechnology Centre in the Silesian University of Technology and at the Academic Computer Center in Gdańsk.

BIBLIOGRAPHY

1. Ciura M., and Deorowicz S.: Experimental Study of Finite Automata Storing Static Lexicons. [in:] Technical Report, Silesian Technical University, Gliwice 1999.
2. Czech Z. J.: Równoległy algorytm dekompozycji języków skończonych. [in:] Wakulicz-Deja A. (ed.): Systemy wspomagania decyzji. Instytut Informatyki Uniwersytetu Śląskiego, Sosnowiec 2010, p. 289÷295.
3. de la Higuera C.: A bibliographical study of grammatical inference. Pattern Recognition, vol. 38 is. 9, 2005, p. 1332÷1348.
4. Mateescu A., Salomaa A., and Yu S.: On the Decomposition of Finite Languages. [in:] Technical Report, Turku Centre for Computer Science, 1998.
5. Salomaa A., Salomaa K., and Yu S.: Length Codes, Products of Languages and Primality. [in:] Martín-Vide C., Otto F., and Fernau H. (eds.): Language and Automata Theory and Applications. LNCS 5196, Springer, Berlin 2008, p. 476÷486.
6. Salomaa A., Yu S.: On the Decomposition of Finite Languages. [in:] Rozenberg G., and Thomas W. (eds.): Developments in Language Theory: Foundations, Applications and Perspectives, World Scientific Publishing, Singapore 2000, p. 22÷31.

7. Watson B.W.: A Fast and Simple Algorithm for Constructing Minimal Acyclic Deterministic Finite Automata. Journal of Universal Computer Science, vol. 8 no. 2, 2002, p. 363÷367.
8. Wieczorek W.: A Local Search Algorithm for Grammatical Inference. [in:] Sempere J.M., and García P. (eds.): Grammatical Inference: Theoretical Results and Applications. LNCS 6339, Springer, Berlin 2010, p. 217÷229.
9. Wieczorek W.: An algorithm for the decomposition of finite languages. Logic Journal of the IGPL, vol. 18 is. 3, 2009, p. 355÷366.
10. Wieczorek W.: Metaheuristics for the Decomposition of Finite Languages. [in:] Kłopotek M.A., Przepiórkowski A., Wierzchoń S.T., and Trojanowski K. (eds.): Recent Advances in Intelligent Information Systems, Akademicka Oficyna Wydawnicza EXIT, 2009, p. 495÷505.
11. Yu S.: Regular languages. [in:] Rozenberg G., and Salomaa A. (eds.): Handbook of Formal Languages: Volume 1., Word, Language, Grammar. Springer, 1997.

Omówienie

Problem dekompozycji języków skończonych jest rozstrzygalny, jednak jego rozwiązanie uznawane jest za trudne. Problem polega na wyznaczeniu pary niepustych języków skończonych L_1, L_2, których złożenie daje w rezultacie język początkowy. Dla danego języka może istnieć zero lub więcej dekompozycji, przy czym języki niemające jakiejkolwiek dekompozycji (poza dekompozycją trywialną) zwane są pierwszymi. Z uwagi na brak uniwersalnego algorytmu, pozwalającego na wyznaczenie zbioru dekompozycji dla danego języka, wykorzystano algorytm przeszukiwania wyczerpującego z obcinaniem przestrzeni rozwiązań. W artykule zaprezentowano istniejące algorytmy zarówno sekwencyjne (rys. 1), jak i równoległe (rys. 2 i 3), rozwiązywania problemu dekompozycji języków skończonych. Stanowiły one podstawę do opracowania ulepszonego algorytmu równoległego (rys. 4). Zaproponowano algorytm minimalizujący liczbę transmisji danych między procesami oraz dokonano oceny efektywności opracowanego algorytmu na podstawie pomiarów czasu wykonania dla implementacji korzystającej z biblioteki MPI. Uzyskane rezultaty (tabela 2) oraz wyznaczone przyspieszenia (tabela 3) dla wybranych języków skończonych umożliwiły ocenę algorytmu jako nie w pełni zadowalającego, w kontekście wykorzystanej liczby procesów. Zaproponowano także dalsze możliwości poprawy działania algorytmu.

Addresses

Tomasz JASTRZĄB: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16
44-100 Gliwice, Polska, Tomasz.Jastrzab@polsl.pl,

Zbigniew J. CZECH: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16
44-100 Gliwice, Polska, Zbigniew.Czech@polsl.pl.