Tomasz JASTRZĄB
INSTITUTE OF INFORMATICS, SILESIAN UNIVERSITY OF TECHNOLOGY
Akademicka Street 16, 44-100 Gliwice, Poland

A comparative analysis of two parallel algorithms for finite language decomposition

M.Sc. Tomasz JASTRZĄB

The author received M.Sc. degree with major in Informatics at the Silesian University of Technology in Gliwice (2011) and is currently a PhD student in the Institute of Informatics. He also works as a Java EE developer at Awteco Sp. z o. o. His research areas involve parallel algorithms for NP-hard problems, and recently the analysis of proteomics spectra.

e-mail: Tomasz.Jastrzab@polsl.pl

Abstract

A finite language is said to possess a non-trivial decomposition if it can be represented as a catenation of two non-empty languages. In this paper two parallel versions of a known sequential algorithm for finding the decomposition of finite languages are proposed. The effectiveness of the algorithms is estimated based on the experimental results obtained for several sample languages.

Keywords: parallel algorithms, finite languages, finite languages decomposition.

Analiza porównawcza dwóch równoległych algorytmów dekompozycji języków skończonych

Streszczenie

Problem dekompozycji języków skończonych jest rozstrzygalny, chociaż trudny do rozwiązania. Problem sprowadza się do wyznaczenia pary niepustych języków skończonych L_1, L_2, takich że w wyniku operacji ich złożenia powstaje język wejściowy. Każdy język skończony posiada dekompozycję trywialną, a oprócz niej zero lub więcej dekompozycji nietrywialnych. Ze względu na brak algorytmu pozwalającego na wyznaczenie zbioru dekompozycji dla dowolnego języka, w artykule zaproponowano rozwiązanie oparte na przeszukiwaniu wyczerpującym z obcinaniem przestrzeni rozwiązań. Artykuł przedstawia dotychczasowe próby rozwiązywania problemu dekompozycji języków skończonych z wykorzystaniem algorytmów sekwencyjnych (rys. 1) oraz równoległych (rys. 2 i 3). Na podstawie znanych algorytmów opracowano ulepszone wersje algorytmu równoległego (rys. 4 i 5). W zaproponowanym rozwiązaniu skoncentrowano się na minimalizacji narzutu czasowego wynikającego z komunikacji pomiędzy procesami. Dokonana ocena efektywności opracowanych algorytmów oparta została o pomiary czasu wykonania dla implementacji z użyciem biblioteki MPI. Uzyskane wyniki (tab. 1), a w szczególności przyspieszenia pozwalają na ocenę rozwiązań jako nie w pełni zadowalających, w odniesieniu do wykorzystanej liczby procesorów.

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

1. Introduction

A language is finite if it is built of a finite set of words over a certain alphabet. As shown in [1] each finite language is regular, thus it may be represented by a finite automaton [2]. As a consequence, finite languages may be applied in the fields of pattern matching, spell-checking [2] and computational biology [3]. Grammatical inference is another area of application of finite languages and their decompositions [4, 5].

The existence of a non-trivial decomposition of a finite language is an intractable [6], yet decidable problem [1]. However, the NP-hardness of this problem has been left as an open issue [6]. Since a universal algorithm for the decomposition of a finite language of arbitrary length is not known, the only way of finding the solution is by using the exhaustive search algorithms [6]. Alternative approaches based on metaheuristics have proven to be unsatisfactory in terms of correctness [7]. Each decomposable language has at least one non-trivial decomposition, although this decomposition may not be unique. Moreover, the decomposition is frequently noncommutative [6, 8], which may cause certain difficulties [9]. An indecomposable language is called prime, by analogy to prime numbers in number theory.

The aim of this paper is to propose and discuss two parallel algorithms for the decomposition of finite languages. The crucial aspect of these algorithms is that they do not focus on a statement about language decomposability, but try to find all possible decompositions of the given language. As an outcome of this process, decomposition set(s) and corresponding factor languages are produced. Both algorithms are assessed experimentally and compared with a sequential version of the algorithm in terms of their execution times. For this purpose, sample languages that are considered hard to decompose were selected. In this context a language is considered hard to decompose if the space of possible solutions cannot be searched in a reasonable time.

The paper is organized into five sections. The second section describes the basic terms and notions related to finite languages and their decompositions. Section 3 is devoted to discussion of existing algorithms, concluding with a proposal of two modified parallel versions of the basic sequential algorithm. In Section 4, sample languages with experimental results of their decomposition are introduced and commented. Section 5 contains final conclusions and remarks.

2. Basic terms and notions

An alphabet, denoted by Σ is defined as a non-empty set of symbols, used for constructing words w of a language L. A set of all possible words generated over the given alphabet is denoted by Σ^*. The number of symbols in a word w, denoted as $|w|$ is called the length of a word. In particular, the length of a word can be equal to 0 for an empty word λ. The cardinality of a language $|L|$ is measured as the number of words in language L. Given words $u, v, w \in \Sigma^*$, a prefix (respectively a suffix) of a word w is defined as a word v such that $w = vu$ (resp. $w = uv$). A prefix (resp. suffix) is *proper* if $v \neq \lambda$ holds.

An operation of catenation can be performed on two words $u, v \in \Sigma^*$, producing a word $w = uv$, $w \in \Sigma^*$, such that it consists of a copy of word u followed by a copy of word v. Similarly, a catenation of two sets of words $U, V \subset \Sigma^*$, produces a set $W = UV$, such that $\{w = uv | w \in \Sigma^* \wedge u \in U \wedge v \in V\}$. *Left* (resp. *right*) *quotient* of a set of words $W \subset \Sigma^*$, given a word $u \in \Sigma^*$, is defined as $u^{-1}W = \{w | uw \in W\}$ (resp. $Wu^{-1} = \{w | wu \in W\}$).

A finite language L may be represented by a minimal acyclic deterministic finite automaton (MADFA in short), defined as a quintuple $\{Q, \Sigma, \delta, s, F\}$, where:
- Q is a *finite* set of states, which follows from the acyclic nature of the automaton,
- Σ is an alphabet,
- $\delta: Q \times \Sigma \to Q$ is a not necessarily *total* [2] transition function, which means that it does not have to be defined for all possible elements of $Q \times \Sigma$,
- $s \in Q$ is the only starting state of the automaton, which is a consequence of automaton's determinism [3], and
- $F \subseteq Q$ is a set of final states.

The fact that the automaton is *minimal* means that for all $p, q \in Q, p \neq q$, it holds that the sets of paths from p or q to states in set F are not equal [3]. A state q will be called *significant* if it satisfies one of the following requirements: it is either non-final and possesses at least two outgoing transitions, or is final with at least one outgoing path [1].

For a language L, the problem of its decomposition may be defined as a problem of finding two non-trivial languages L_1, L_2, such that:

$$L = L_1 L_2. \quad (1)$$

Alternatively it may be stated as the problem of finding a non-empty subset of states $P \subseteq Q$ of the MADFA accepting language L, satisfying the condition that

$$L = R_1^P R_2^P, \quad (2)$$

where R_1^P and R_2^P are given as:

$$R_1^P = \cup_{p \in P} \overleftarrow{p}, \quad (3)$$

$$R_2^P = \cap_{p \in P} \overrightarrow{p}, \quad (4)$$

where symbols $\overleftarrow{p}$ and $\overrightarrow{p}$ denote the left (resp. right) languages for the state p. The *left language* for a state p denotes the set of words $W \subset \Sigma^*$, spelled out on the paths from the starting state s to state p. Respectively, the *right language* for a state p is defined as a set of words $W \subset \Sigma^*$, produced by following the paths from state p to the final states in set F. It is shown in [8], that for L_i, R_i^P from (1) and (2), the following is true $L_i \subseteq R_i^P$, $i = 1, 2$.

3. The algorithms

3.1. Sequential algorithm

In [1] a sequential branch-and-bound algorithm for solving the problem of finite language decomposition is proposed. The initial version searched only for the first decomposition, thus it was modified to enable finding all available decompositions of a language L. The pseudo-code of the core part of the algorithm is shown in fig. 1. It assumes that a minimal acyclic deterministic finite automaton was already built beforehand. In this particular case MADFA is built based on the algorithm given in [5].

```
    function decompose(I, D)
       // at or below cut-off level
1:     if |s(I) - D| <= K then
          // power set generation
2:        for each A ⊆ (s(I) − D) do
             // D complemented with A
3:           set P = D ∪ A
             // based on (2), (3), (4)
4:           if P is a solution then
5:              print P, L1, L2
6:           end if
7:        end for
8:     else
9:        sort I according to |s(I[i])|
10:       if |s(I[0])| = 0 then    // I is empty
11:          return
12:       else
             // word and its states removed
13:          remove I[0] from I
14:          for each q ∈ s(I[0]) do
15:             set s = suffix of w at q
                // states not generating s removed
16:             remove states based on s
                // D extended with q
17:             decompose(I, D ∪ q)
18:          end for
19:       end if
20:  end if
```

Fig. 1. Sequential algorithm for the decomposition of finite languages
Rys. 1. Algorytm sekwencyjny dekompozycji języków skończonych

In Fig. 1, variable I denotes the set of input pairs (w, *s(w)*), where w is a word in L, and *s(w)* is a set of significant states of w. At the beginning I contains all words in L, and consequently the sum of *s(w)* for all w, gives the whole set of significant states. Variable D, denotes the current decomposition set, which is initially empty, and is successively updated in line 17. The constant K denotes the cut-off level, and is a parameter of the algorithm. There are two phases of the algorithm, one responsible for generation and verification of potential decomposition sets (lines 2-7) and the other responsible for pruning the solution space (lines 15-16). The pruning is based on the conclusion drawn from (4). Namely, since for the decomposition to be valid every word w in L, has to be split in two parts by at least one state q in D, it follows that the states that do not have a common suffix with q (or generally with states in D), cannot be added to the decomposition set. In the following sections lines 2-7 will be referred to as *below-K phase*, and lines 9-19 as *above-K phase*.

3.2. Parallel algorithms

One of the parallel algorithms solving the problem of finite language decomposition is presented in [10]. It was based on the observation that certain amount of time is spent on the tasks executed in below-K phase of the algorithm presented in Fig. 1. Moreover, it was pointed out that the investigation of the respective subsets of the power set could be performed independently. The master-workers scheme of cooperation between processes was used in this approach. To reduce the amount of inter-process communication, buffering was introduced in the master process.

The pseudo-codes of both master and worker processes are shown in Figs. 2 and , respectively. As shown in lines 4-7 in Fig. 2, the tasks are assigned to worker processes according to the round robin algorithm. As mentioned before, buffering is performed in the master process (lines 2-3), with b_size constant denoting the amount of data sent at once. The sent data packets consist of the pairs of sets of states *s(I)* and *D* as shown in line 2 in Fig. 2.

```
    function decomposeM(I, D)
       // at or below cut-off level
1:     if |s(I) - D| <= K then
2:        put pair (s(I), D) into buffer
3:        b_count = b_count + 1
          // sending point reached
4:        if b_count = b_size then
5:           send buffer in round robin fashion
6:           b_count = 0
7:        end if
8:     else
9:        above-K phase as defined in fig. 1
          // some pairs not sent yet
10:       if b_count != 0 then
11:          send remaining pairs
12:       end if
13:       send finish signal // master finished
14:    end if
```

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

```
    function decomposeW()
1:     while true do
          // pairs (s(I), D) or finish
2:        receive buffer from master
          // finish signal received
3:        if buffer = finish signal then
4:           break // worker finished
5:        else
6:           for each pair Pr in buffer do
7:              below K-phase as defined in fig. 1
                with Pr.s(I), Pr.D
8:           end for
9:        end if
10:    end while
```

Fig. 3. Worker process in a parallel algorithm
Rys. 3. Proces wykonawczy w algorytmie równoległym

The approach proposed in [10] was assessed as not satisfactory in terms of speedup values. Two main reasons were distinguished, being the overhead induced by communication between processes and unequal load distribution. Thus, alternative parallel algorithms presented further, were aimed at improving these elements.

Basing on the sequential algorithm presented in Fig. 1 and conclusions drawn from the parallel algorithm given in Figs. 2 and 3, two alternative solutions were considered. Their main goal was to reduce the amount of inter-process communication, which was achieved by allowing all the processes to execute concurrently the same code. However, some branching was also introduced in the independent parts of processing, to speed up computations. The algorithms discussed below aim at two different phases, below-K phase, already shown as being the source of parallelism, and above-K phase which also allows for independent execution. For ease of reference, the approach working in below-K phase will be denoted shortly as BK, with the other called AK.

The BK algorithm is depicted in Fig. 4, where c_*no* variable is used for implementing every *n*-th iteration approach. In this solution processes execute the main *decompose* function concurrently and they branch into separate execution paths in below-K phase verifying only selected decomposition sets (lines 3-8 in Fig. 4). It is easy to verify that all potential decomposition sets are still covered in this approach, because *c_no* is modified modulo *p_cnt*, where *p_cnt* is the number of processors used (line 3). Variable *p_no* denotes the process rank.

```
    function decompose(I, D)
       // at or below cut-off level
1:     if |s(I) - D| <= K then
          // power set generation
2:        for each A ⊆ (s(I) − D) do
             // combination number modulo
             // processors count must match
             // process rank
3:           if (c_no++ % p_cnt) = p_no then
                // D complemented with A
4:              set P = D ∪ A
                // based on (2), (3), (4)
5:              if P is a solution then
6:                 print P, L1, L2
7:              end if
8:           end if
9:        end for
10:    else
11:       above-K phase as defined in fig. 1
12:    end if
```

Fig. 4. A parallel algorithm (BK version)
Rys. 4. Algorytm równoległy (wersja BK)

In the AK approach, the emphasis was put on concurrent execution of the loop traversing states in the word chosen after sorting the words according to the number of their significant states. Such an approach was justified by the fact that these states represent independent search paths. At first a straightforward method of processors assignment was taken, namely states were assigned to processors in round robin fashion at recursion level 1. However, usually there were not enough states to assign to each process. Thus a more advanced approach was taken.

Taking into account that the number of states in a selected word was usually low, some states had to be analyzed by several processes, so it seemed justified to combine AK and BK solutions (AK-BK algorithm). Two assignment schemes were designed. Namely, either equally sized groups of processes were devoted to each state, or group size was based on the frequency of suffix appearance. The second method was based on the assumption that the suffix appearing in more words will have a smaller chance of pruning incorrect states. Consequently the search path in such a case could be longer and more processes might be needed to make the execution faster. In both cases the assignment of processors was performed at the first recursion level. The general pseudo-code of AK-BK algorithm is shown in Fig. 5. Line 3 in Fig. 5 has similar meaning as line 3 in Fig. 4, but this time combination numbers (*c_no*) are considered separately for each group of processes. Lines 15-17 use one of the assignment algorithms described above, while in line 20, the decision is made depending on the recursion level. For the first level the states are assigned to successive groups of processes, for the other levels the algorithm works in the way described for the sequential algorithm, but with more processes at once. This way several paths are explored concurrently, and below-K execution is also performed in parallel.

```
    function decompose(I, D)
       // at or below cut-off level
1:     if |s(I) - D| <= K then
          // power set generation
2:        for each A ⊆ (s(I) − D) do
             // combination number must match
             // process rank
3:           if (c_no++ for p_no) then
                // D complemented with A
4:              set P = D ∪ A
                // based on (2), (3), (4)
5:              if P is a solution then
6:                 print P, L1, L2
7:              end if
8:           end if
9:        end for
10:    else
11:       sort I according to |s(I[i])|
12:       if |s(I[0])| = 0 then    // I is empty
13:          return
14:       else
15:          if level = 1 then
16:             assign states to processors
17:          end if
             // word and its states removed
18:          remove I[0] from I
19:          for each q ∈ s(I[0]) do
20:             if (q for p_no) then
21:                set s = suffix of w at q
              // states not generating s removed
22:                remove states based on s
                   // D extended with q
23:                decompose(I, D ∪ q)
24:             end if
25:          end for
26:       end if
27:    end if
```

Fig. 5. A parallel algorithm (AK-BK version)
Rys. 5. Algorytm równoległy (wersja AK-BK)

4. The experiments

The experiments were conducted for four decomposable languages of different cardinality. They had 800, 5317, 6034 and 6583 words, and were generated over the following alphabets $\Sigma_1 = \{a, b\}$ (language with 800 words) and $\Sigma_2 = \{a, b, c, d\}$ (other three languages). The number of significant states for each MADFA was equal to 17, 73, 67, and 74, respectively.

The algorithms were implemented in C language using Message Passing Interface (MPI). The interpreter run on Intel Xeon X5650 2,66 GHz processors, with the nodes interconnected with Infiniband 40 Gb/s network. The operating system was Red Hat Enterprise Linux 5.

4.1. Experimental results

According to the author's best knowledge there are no benchmarks for the problem of finite language decomposition. Since out of the three approaches (AK, BK and AK-BK) only the second one provided relatively satisfactory results, its execution times are shown in Tab. 1, and the remaining results are only commented upon. The times provided in Tab. 1 are expressed in seconds, and were measured using *clock()* function. They represent execution times of the function shown in Fig. 4. The

experiments were performed for cut-off level parameter K in the range [10, 15]. Tab. 1 reports the results obtained for K chosen to provide the shortest execution times (actual K values are given in column headers for each language separately). The number of processors used is given in column p_cnt. The values presented in Tab. 1 are the average values out of three measurements performed for each language and processor count. The last row represents the best speedup value obtained for each language, which was computed using formula:

$$S_{p_cnt} = \frac{T(1)}{T(p_cnt)}, \quad (5)$$

where $T(1)$ and $T(p_cnt)$ represent execution times for 1 and p_cnt processors, respectively.

Tab. 1. Execution times for parallel algorithm in BK-version
Tab. 1. Czasy wykonania algorytmu równoległego w wersji BK

p_cnt	800 words (K = 15)	5317 words (K = 14)	6034 words (K = 10)	6583 words (K = 15)
1	77.55	2274.95	248.01	3192.74
2	50.78	1280.73	136.46	1623.63
4	24.03	726.48	93.57	895.49
6	17.77	541.28	69.79	612.83
8	14.89	447.65	62.09	471.93
10	8.55	393.60	38.95	386.74
S_{10}	**9.07**	**5.78**	**6.37**	**8.26**

As it can be observed based on the results presented in Tab. 1 the speedup values vary between languages although the overall trend is quite satisfactory. A detailed investigation of the execution times of the below-K phase reveals that this part of algorithm is parallelized relatively well, which may suggest that load distribution in this part is well-balanced. However, since the results for various languages differ, it can be concluded that the load distribution is much language-dependent. Thus, the use of more processors will not always guarantee better results.

More importantly, K values reported in Tab. 1 do not always correspond to the best speedup values obtained. As an example of such a situation, the case of language with 6034 words can be given, which for K = 13 gives speedup of 9.67 for 10 processes, although the overall execution time is worse than the one for K equal to 10 and only 2 processors. Nevertheless, the characteristic feature appearing in all the languages is that if the processing time of below-K phase is approximately equal to the total execution time of the decomposition function, the speedup values can even reach the levels close to linear.

The results obtained for pure AK approach were unsatisfactory, due to the way the successive iterations were performed, namely that the word with minimum number of states was chosen. As said in Section 3, most processes did not perform useful work, due to the small number of states to be analyzed, thus increase in the number of processors did not produce better a performance.

In the case of AK-BK approach the below-K phase was again parallelized well. However, since some of the processes were assigned to different exploration paths during the first recursion level, the overall results were worse compared to BK approach. Furthermore, large differences in the amount of work performed by each process were noticed. Such a situation was observed for both assignment schemes (equally-sized and suffix-frequency-based), which led to more thorough investigation of the exploration tree structure.

This research allowed observing that the problem in efficient parallelization of the above-K phase is that the tree usually contains a single dominating path. It means that the exploration time of this path is much longer than that of the other paths. For instance, for the language with 6034 words, the relation of the main path execution time to the other times was like 12200 : 175 : 1, and in other cases these proportions were even more spectacular. The only exception from this rule was the language with 800 words, where the differences in execution times of various paths were not that large. Consequently, it may be again concluded, that it is a language-dependent feature influenced heavily by K value choice.

5. Conclusions

The paper investigated two proposals of parallel algorithms for the problem of finite language decomposition. The main aim was to improve the performance of finding all possible decomposition sets for a given language. The algorithms were based on the sequential branch-and-bound algorithm. They were assessed by measurements of the execution times, which allowed observing that further improvements are required.

The enhancements may involve better load balancing of the parallel part, so as to keep all processes approximately equally occupied. Since the cut-off level value turned out to be a crucial parameter of the algorithms, an adaptive approach to its adjustment may be considered as an improvement option. As for the parallelism in the above-K phase, changing the way of selection of successive words to follow the search tree structure may be also helpful.

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.

The author would also like to acknowledge the help and useful remarks of prof. Zbigniew Czech related to BK version of the algorithm.

6. References

[1] Wieczorek W.: An algorithm for the decomposition of finite languages. Logic Journal of the IGPL, vol. 18 is. 3, pp. 355÷366, 2009.

[2] Yu S.: Regular languages. [in:] Rozenberg G., and Salomaa A. (eds.): Handbook of Formal Languages: Volume 1., Word, Language, Grammar. Springer, 1997.

[3] Watson B.W.: A Fast and Simple Algorithm for Constructing Minimal Acyclic Deterministic Finite Automata. Journal of Universal Computer Science, vol. 8 no. 2, pp. 363÷367, 2002.

[4] de la Higuera C.: A bibliographical study of grammatical inference. Pattern Recognition, vol. 38 is. 9, pp. 1332÷1348, 2005.

[5] 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, pp. 217÷229, 2010.

[6] Mateescu A., Salomaa A., and Yu S.: On the Decomposition of Finite Languages. [in:] Technical Report, Turku Centre for Computer Science, 1998.

[7] 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, pp. 495÷505, 2009.

[8] 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, pp. 22-31, 2000.

[9] 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, pp. 476÷486, 2008.

[10] Czech Z.J.: Równoległy algorytm dekompozycji języków skończonych. [in:] Wakulicz-Deja A. (ed.): Systemy wspomagania decyzji. Instytut Informatyki Uniwersytetu Śląskiego, Sosnowiec, pp. 289÷295, 2010.

otrzymano / received: 12.03.2014
przyjęto do druku / accepted: 02.05.2014 *artykuł recenzowany / revised paper*