HAL open science

Parallelising the Computation of Minimal Absent Words

Carl Barton, Alice Héliou, Laurent Mouchard, Solon P. Pissis

To cite this version:

Carl Barton, Alice Héliou, Laurent Mouchard, Solon P. Pissis. Parallelising the Computation of Minimal Absent Words. PPAM 2015, Sep 2015, Cracovie, Poland. 10.1007/978-3-319-32152-3_23 . hal-01255489

HAL Id: hal-01255489

https://hal.science/hal-01255489v1

Submitted on 13 Jan 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Parallelising the Computation of Minimal Absent Words

Carl Barton[1], Alice Heliou[2,3], Laurent Mouchard[4], and Solon P. Pissis[5]

[1] The Blizard Institute, Barts and The London School of Medicine and Dentistry, Queen Mary University of London, UK
c.barton@qmul.ac.uk
[2] Inria Saclay-Île de France, AMIB, Bâtiment Alan Turing, France
[3] Laboratoire d'Informatique de l'École Polytechnique (LIX), CNRS UMR 7161, France
alice.heliou@polytechnique.org
[4] University of Rouen, LITIS EA 4108, TIBS, Rouen, France
laurent.mouchard@univ-rouen.fr
[5] Department of Informatics, King's College London, London, UK
solon.pissis@kcl.ac.uk

Abstract. An *absent word* of a word y of length n is a word that does not occur in y. It is a *minimal absent word* if all its proper factors occur in y. Minimal absent words have been computed in genomes of organisms from all domains of life; their computation also provides a fast alternative for measuring approximation in sequence comparison. There exists an $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space algorithm for computing all minimal absent words on a fixed-sized alphabet based on the construction of suffix array (Barton *et al.*, 2014). An implementation of this algorithm was also provided by the authors and is currently the fastest available. In this article, we present a new $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space algorithm for computing all minimal absent words; it has the desirable property that, given the indexing data structure at hand, the computation of minimal absent words can be executed in parallel. Experimental results show that a multiprocessing implementation of this algorithm can accelerate the overall computation by more than a factor of two compared to state-of-the-art approaches. By excluding the indexing data structure construction time, we show that the implementation achieves near-optimal speed-ups.

Keywords: algorithms on strings, absent words, suffix array

1 Introduction

Sequence comparison is an important step in many tasks in bioinformatics. It is fundamental in many applications; from phylogenies reconstruction to the reconstruction of genomes. Traditional algorithms for measuring approximation in sequence comparison are based on the notions of distance or of similarity between sequences, which are generally computed through sequence alignment

techniques. An issue with using alignment techniques is that they are computationally expensive, requiring quadratic time in the length of the sequences—a truly sub-quadratic algorithm for this problem seems highly unlikely [1]. This has led to increased research into *alignment free* techniques [10].

Whole-genome alignments prove computationally intensive and have little biological significance. Hence standard notions for sequence comparison are gradually being complemented and in some cases replaced by alternative ones that refer either implicitly or explicitly to the composition of sequences in terms of their constituent patterns. One such notion is based on comparing the words that are absent in each sequence. A word is an *absent word* of some sequence if it does not occur in the sequence. Absent words represent a type of *negative information*: information about what does not occur in the sequence. For instance, considering the words which occur in one sequence but do not in another can be used to detect mutations or other biologically significant events [17].

Given a sequence of length n, the number of absent words of length at most n is exponential in n. However, the number of certain classes of absent words is only linear in n. A *minimal absent word* of a sequence is an absent word whose proper factors all occur in the sequence. Notice that minimal and *shortest absent words* [18] are not the same; minimal absent words are a superset of shortest absent words [15]. An upper bound on the number of minimal absent words is known to be $\mathcal{O}(\sigma n)$ [6, 13], where σ is the size of the alphabet. This suggests that it may be possible to compare sequences in time proportional to their lengths, for a fixed-sized alphabet, instead of proportional to the product of their lengths [10].

Recently, there has been a number of studies on the biological significance of absent words in various species. The most comprehensive study on the significance of absent words is probably [2]; in this, the authors suggest that the deficit of certain subsets of absent words in vertebrates may be explained by the hypermutability of the genome. It was later found in [9] that the compositional biases observed in vertebrates in [2] are not uniform throughout different sets of minimal absent words. Moreover, the analyses in [9] support the hypothesis that minimal absent words are inherited through a common ancestor, in addition to lineage-specific inheritance, only in vertebrates. In [8], the minimal absent words in four human genomes were computed, and it was shown that, as expected, intra-species variations in minimal absent words were lower than inter-species variations. Very recently, in [17], it was shown that there exist three minimal words in the *Ebola* virus genomes which are absent from human genome. The authors suggest that the identification of such species-specific sequences may prove to be useful for the development of both diagnosis and therapeutics.

From an algorithmic perspective, an $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space algorithm for computing all minimal absent words on a fixed-sized alphabet based on the construction of suffix automata was presented in [6]. An alternative $\mathcal{O}(n)$-time solution for finding minimal absent words of length at most ℓ, such that $\ell = \mathcal{O}(1)$, based on the construction of tries of bounded-length factors was presented in [5]. A drawback of these approaches, in practical terms, is that the construction of

suffix automata (or of tries) often have a large memory footprint. Hence, an important problem was to be able to compute minimal absent words with more memory-efficient data structures (cf. [4]).

The computation of minimal absent words based on the construction of suffix arrays was considered in [15]; although this algorithm has a linear-time performance in practice, the worst-case time complexity is $\mathcal{O}(n^2)$. The first $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space suffix-array-based algorithm was recently presented in [3] to bridge this unpleasant gap. An implementation of this algorithm is currently, and to the best of our knowledge, the fastest available for the computation of minimal absent words. With the continuous efforts in whole-genome sequencing, the computation of minimal absent words remains the main bottleneck in analysing a large set of large genomes [8, 9, 17]. Hence due to the large amounts of data being produced, it is desirable to further engineer this computation.

Our Contribution. In this article, our contribution is threefold: (a) We present a new $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space algorithm for computing all minimal absent words on a fixed-sized alphabet; (b) We show that this algorithm has the desirable property that, given the relevant indexing data structure at hand, the computation of minimal absent words can be executed in parallel; and (c) We make available an implementation of this algorithm for shared-memory multiprocessing programming. Experimental results, using real and synthetic data, show that the *overall* computation is accelerated by more than a factor of two compared to the state of the art. By excluding the indexing data structure construction time, we show that the implementation achieves *near-optimal* speed-ups. This is important as engineering further the involved indexing data structure construction is an ongoing research topic [16], which is beyond the scope of this article.

2 Definitions and Notation

To provide an overview of our result and algorithm, we begin with a few definitions from [3]. Let $y = y[0]y[1]\,..\,y[n-1]$ be a *word* of *length* $n = |y|$ over a finite ordered *alphabet* Σ of size $\sigma = |\Sigma| = \mathcal{O}(1)$. We denote by $y[i\,..\,j] = y[i]\,..\,y[j]$ the *factor* of y that starts at position i and ends at position j and by ε the *empty word*, word of length 0. We recall that a prefix of y is a factor that starts at position 0 ($y[0\,..\,j]$) and a suffix is a factor that ends at position $n-1$ ($y[i\,..\,n-1]$), and that a factor of y is a *proper* factor if it is not the empty word or y itself.

Let x be a word of length $0 < m \leq n$. We say that there exists an *occurrence* of x in y, or, more simply, that x *occurs in* y, when x is a factor of y. Every occurrence of x can be characterised by a starting position in y. Thus we say that x occurs at the *starting position* i in y when $x = y[i\,..\,i+m-1]$. Opposingly, we say that the word x is an *absent word* of y if it does not occur in y. The absent word x, $m \geq 2$, of y is *minimal* if and only if all its proper factors occur in y.

We denote by SA the *suffix array* of y, that is the array of length n of the starting positions of all sorted suffixes of y, i.e. for all $1 \leq r < n$, we have $y[\mathsf{SA}[r-1]\,..\,n-1] < y[\mathsf{SA}[r]\,..\,n-1]$ [12]. Let $\mathsf{lcp}(r, s)$ denote the length of the

longest common prefix of the words $y[\mathsf{SA}[r]\,..\,n-1]$ and $y[\mathsf{SA}[s]\,..\,n-1]$, for all $0 \leq r, s < n$, and 0 otherwise. We denote by LCP the *longest common prefix* array of y defined by $\mathsf{LCP}[r] = \mathsf{lcp}(r-1, r)$, for all $1 \leq r < n$, and $\mathsf{LCP}[0] = 0$. SA [14] and LCP [7] of y can be computed in time and space $\mathcal{O}(n)$.

In this article, we consider the following problem.

MinimalAbsentWords **Input:** a word y on Σ of length n **Output:** all tuples $< a, (i, j) >$, such that word x, defined by $x[0] = a$, $a \in \Sigma$, and $x[1\,..\,m-1] = y[i\,..\,j]$, $m \geq 2$, is a minimal absent word of y

3 Algorithm pMAW

In this section, we present algorithm pMAW, a new $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space algorithm for computing all minimal absent words of a word of length n using arrays SA and LCP. We first start by explaining some useful properties from [15] we use in algorithm pMAW. Then we present our algorithm in detail, and, finally, we show how it can be adapted for parallel computing.

3.1 Useful Properties

A minimal absent word $x[0\,..\,m-1]$ of a word $y[0\,..\,n-1]$ is an absent word whose proper factors all occur in y; *equivalently*, both the longest proper suffix and prefix of x occur in y.

Definition 1. *A* repeated pair *in a word* $\boldsymbol{y}$ *is a tuple* $< \boldsymbol{i}, \boldsymbol{j}, \boldsymbol{w} >$ *such that word* $\boldsymbol{w}$ *occurs in* $\boldsymbol{y}$ *at starting positions* $\boldsymbol{i}$ *and* $\boldsymbol{j}$*. A repeated pair is* right *(resp.* left*)* maximal*, if* $\boldsymbol{y}[\boldsymbol{i} + |\boldsymbol{w}|] \neq \boldsymbol{y}[\boldsymbol{j} + |\boldsymbol{w}|]$ *(resp.* $\boldsymbol{y}[\boldsymbol{i} - 1] \neq \boldsymbol{y}[\boldsymbol{j} - 1]$*) A repeated pair is* maximal *if it is left maximal and right maximal.*

Lemma 2 ([15]). *If* $\boldsymbol{awb}$ *is a minimal absent word of a word* $\boldsymbol{y}$*, where* $\boldsymbol{a}$ *and* $\boldsymbol{b}$ *are letters and* $\boldsymbol{w}$ *a word, then there exist two positions* $\boldsymbol{i}$ *and* $\boldsymbol{j}$ *such that* $< \boldsymbol{i}, \boldsymbol{j}, \boldsymbol{w} >$ *is a maximal repeated pair in* $\boldsymbol{y}$*.*

By Lemma 2, we can exhaustively compute minimal absent words by examining all the maximal repeated pairs. To compute maximal repeated pairs, we consider all right maximal repeated pairs and check the letters that occur just before.

Definition 3. *Given the* LCP *array of a word of length* $\boldsymbol{n}$*, we say that interval* $[\boldsymbol{i}, \boldsymbol{j}]$*,* $0 \leq \boldsymbol{i} < \boldsymbol{j} \leq \boldsymbol{n} - 1$*, is an* LCP-interval *of* LCP-depth $\boldsymbol{d}$ *if*

- $\mathsf{LCP}[\boldsymbol{i}] < \boldsymbol{d}$, and $\boldsymbol{j} = \boldsymbol{n} - 1$ or $\mathsf{LCP}[\boldsymbol{j} + 1] < \boldsymbol{d}$
- $\mathsf{LCP}[\boldsymbol{k}] \geq \boldsymbol{d}$, for all $\boldsymbol{i} < \boldsymbol{k} \leq \boldsymbol{j}$
- $\mathsf{LCP}[\boldsymbol{k}] = \boldsymbol{d}$, for at least one $\boldsymbol{k}, \boldsymbol{i} < \boldsymbol{k} \leq \boldsymbol{j}$.

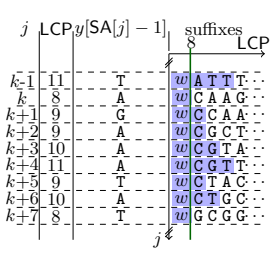
Right maximal repeated pairs are given by the suffix array with the notion of LCP-interval. Indeed if positions i and j are in an LCP-interval of depth d then $< i, j, y[\mathsf{SA}[i]\,..\,\mathsf{SA}[i] + d - 1] >$ is a right maximal repeated pair. Analogously, if $< i, j, w >$ is a right maximal repeated pair then i and j are in the same LCP-interval of depth $|w|$.

3.2 Computation of Minimal Absent Words

For the rest of this section we denote minimal absent words by maws. We first pre-compute SA, LCP, and a bit-vector v such that $v[i] = 1$ if and only if LCP$[i]$ is a local maximum. We use rank and select data structures and denote by MaxRank(k) the operation giving the number of 1's in $[0 : k)$ and by MaxSelect(k) the operation giving the position of the kth 1. The following function presents maws computation for a given interval $[k_1, k_2)$ of SA and LCP.

Function *ComputeMaws* (k_1, k_2, y, SA, LCP, MaxRank, MaxSelect)
  SetLetter$\leftarrow\emptyset$; LifoPos.push(0); LifoSet.push(SetLetter);
  foreach $t \in [\text{MaxRank}(k_1) + 1 : \text{MaxRank}(k_2)]$ **do**
    $i\leftarrow$MaxSelect(t); *left*$\leftarrow i - 1$; *right*$\leftarrow i + 1$;
    pos$\leftarrow$LifoPos.top(); *lpos*$\leftarrow$LCP[*pos*]; SetLetter$\leftarrow\emptyset$;
    while *1* **do**
      while *pos* > 0 and LCP$[i] <$ *lpos* **do**
        we pop from LifoPos the positions with an LCP value equal to *lpos*; we pop their set of letters from LifoSet; we have visited the whole LCP-interval of depth *lpos*, so we infer maws using these sets and SetLetter; we update *left* and *right*; *pos*$\leftarrow$LifoPos.top(); *lpos*$\leftarrow$LCP[*pos*];
      if LCP$[i] > \max($LCP[*left*], LCP[*right*], *lpos*$)$ **then**
        we have visited the whole LCP-interval of depth LCP$[i]$, so we infer maws with SetLetter, $y[$SA$[i]-1]$, and $y[$SA[*left*]$-1]$;
      SetLetter$\leftarrow$SetLetter $\cup \{y[$SA$[i] - 1]\}$;
      if LCP[*left*] $=$ LCP$[i]$ *or* LCP[*right*] $=$ LCP$[i]$ **then**
        LifoPos.push(i); LifoSet.push(SetLetter);
        we push onto LifoPos all the successive neighbours of interval (*left*,*right*) with an LCP value equal to LCP$[i]$; for each of them we push onto LifoSet the letter preceding their corresponding suffix; we update *left* and *right*;
      if LCP[*right*] $\leq$ LCP[*left*] $<$ LCP$[i]$ **then** $i\leftarrow$*left*; *left*$\leftarrow i - 1$;
      else if LCP[*right*] $>$ LCP$[i]$ **then** we push onto stacks the positions skipped and their corresponding set of letters; **break**;
      else $i\leftarrow$*right*; *right*$\leftarrow i + 1$;

If i is a local maximum in the LCP array, then $[i-1, i]$ is the LCP-interval of LCP-depth LCP$[i]$ that contains i. Consequently our idea is to start the computation at the first local maximum of the LCP array and to visit the surrounding positions in decreasing order of their LCP value. In this process we keep in the array SetLetter the set of letters that occur before the repeated factor. When we reach a local minimum we store its position on the SA array in the stack LifoPos, and the current array SetLetter in the stack LifoSet. We will analyse them once we have visited their whole LCP-interval. In this way, we consider each maximal



step	i	*left*	*right*	SetLetter	Inferred maws and action on stacks
1	$k+4$	$k+3$	$k+5$	$\emptyset$	
2	$k+3$	$k+2$	$k+5$	{A}	
3	$k+2$	$k+1$	$k+5$	{A}	we push $k+2$, $k+1$, and $k+5$ onto LifoPos; we push SetLetter,
		k	$k+6$		{G}, and {T} onto LifoSet
4	$k+6$	$k+5$	$k+7$	$\emptyset$	we infer 2 maws: AwCTA, TwCTG
5	$k+5$	$k+4$	$k+7$	{A}	$k+5$ is already in LifoPos
6	$k+7$	$k+4$	$k+8$	{A,T}	we pop $k+5$, $k+1$, and $k+2$ from LifoPos and {T}, {G}, {A} from LifoSet
		k		{A,G,T}	we infer 7 maws: GwCA, TwCA, AwCC, TwCC, GwCG, TwCG, GwCT

Fig. 1: Illustration of the algorithm step by step for the interval $[k, k+7)$. The example is taken from the *Lactobacillus casei* genome (Accession #: NC010999). $w =$ TCTGAGCG is a common prefix of the considered suffixes and $k = 2,554,910$.

repeated pair and infer from them the whole set of maws using Lemma 2. An example of this function is illustrated in Fig. 1. Contrary to MAW [3], the previous linear-time algorithm, in pMAW we do not consider our data structures globally; we rather consider each LCP-interval *independently*. This important property will allow us to use parallel computations, as shown in Section 3.3.

Overall Complexity. We use arrays SA and LCP, which can be computed in time and space $\mathcal{O}(n)$ [14, 7]. There also exists a representation which uses $n + o(n)$ bits of storage space and supports rank and select on a bit-vector of size n in constant time [11]. We also use two stacks, LifoPos and LifoSet, where we push and pop $\mathcal{O}(n)$ elements, each containing at most σ integers. Thus the whole algorithm requires time and space $\mathcal{O}(\sigma n)$. We obtain the following result.

Theorem 4. *Algorithm pMAW solves problem* MinimalAbsentWords *in time and space* $\mathcal{O}(n)$.

The *advantages* of pMAW over existing works are as follows. It is (provably) linear-time in the worst case as opposed to the one in [15]. Contrary to the linear-time algorithm in [3], we explicitly compute the LCP-intervals. For a given depth, LCP-intervals have no overlap, therefore we can consider them independently.

3.3 Parallelisation Scheme

Lemma 5. *Let* $\boldsymbol{y}$ *be a word of length* $\boldsymbol{n}$ *over an alphabet of size* $\boldsymbol{\sigma}$ *and let* $\boldsymbol{\ell}$ *be the length of the shortest minimal absent word of* $\boldsymbol{y}$*. Then the following hold:*

- *For all* $\boldsymbol{k} \in [0,\ \boldsymbol{\ell}-2]$, $|\{\boldsymbol{s} \in [0, \boldsymbol{n}-1] : \mathsf{LCP}[\boldsymbol{s}] = \boldsymbol{k}\}| = (\boldsymbol{\sigma}-1)\boldsymbol{\sigma}^{\boldsymbol{k}} + 1$;
- *For all* $\boldsymbol{k} \in [\boldsymbol{\ell}-1, \boldsymbol{n}-1]$, $|\{\boldsymbol{s} \in [0, \boldsymbol{n}-1] : \mathsf{LCP}[\boldsymbol{s}] = \boldsymbol{k}\}| < (\boldsymbol{\sigma}-1)\boldsymbol{\sigma}^{\boldsymbol{k}} + 1$.

Proof. Let $k \in [0, n-1]$, we denote by $s_0, \ldots, s_{m-1}$, ordered increasingly, the m elements of the set $\{s \in [0, n-1] : \mathsf{LCP}[s] = k\}$. For all $i \in [0, m-1]$, we have $y[\mathsf{SA}[s_i-1] \mathinner{.\,.} \mathsf{SA}[s_i-1]+k-1] = y[\mathsf{SA}[s_i] \mathinner{.\,.} \mathsf{SA}[s_i]+k-1]$ and $y[\mathsf{SA}[s_i-1]+k] < y[\mathsf{SA}[s_i]+k]$. We consider the pair (s_i, s_{i+1}) with $i \in [0, m-2]$, there are two cases:

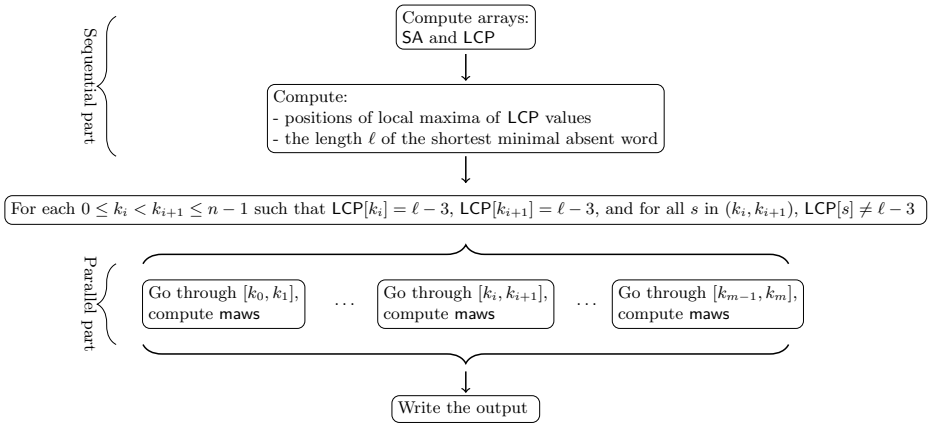


Fig. 2: Overview of Algorithm pMAW

- $\mathsf{lcp}(s_i, s_{i+1}) = k$, so $y[\mathsf{SA}[s_i] \mathinner{..} \mathsf{SA}[s_i] + k - 1] = y[\mathsf{SA}[s_{i+1}] \mathinner{..} \mathsf{SA}[s_{i+1}] + k - 1]$ and $y[\mathsf{SA}[s_i - 1] + k] < y[\mathsf{SA}[s_i] + k] \leq y[\mathsf{SA}[s_{i+1} - 1] + k] < y[\mathsf{SA}[s_{i+1}] + k]$. The alphabet is of size σ; this can happen at most $\sigma - 2$ times consecutively.
- $\mathsf{lcp}(s_i, s_{i+1}) < k$, so $y[\mathsf{SA}[s_i] \mathinner{..} \mathsf{SA}[s_i] + k - 1] < y[\mathsf{SA}[s_{i+1}] \mathinner{..} \mathsf{SA}[s_{i+1}] + k - 1]$. There are σ^k different words of length k; this can happen at most $\sigma^k - 1$ times.

In the first case, we have an additional sub-case, when $\mathsf{SA}[s_i - 1] + k = n$. Then $y[\mathsf{SA}[s_i - 1] + k]$ is not a letter of the alphabet Σ, so we have one more position with an LCP value equal to k. Thus, there are at most $(\sigma - 1)\sigma^k$ pairs (s_i, s_{i+1}), so there are at most $(\sigma - 1)\sigma^k + 1$ positions with an LCP value equal to k.

The equality holds if and only if all the words of length $k + 1$ appear in y, so only if $k < \ell' - 1$ where ℓ' is the length of the shortest absent word. A minimal absent word is an absent word so $\ell \geq \ell'$. Let x be a shortest absent word, then all its proper factors occur in y because they are smaller than x, so x is a minimal absent word. Therefore $\ell = \ell'$, the equality holds if and only if $k \in [0, \ell - 2]$. $\square$

By Lemma 5, the length ℓ of the shortest minimal absent word of some word of length n satisfies: $\ell - 1 = \min\{k \geq 0 : |\{s \in [0, n-1] : \mathsf{LCP}[s] = k\}| < (\sigma - 1)\sigma^k + 1\}$. As the alphabet is of size σ, there are σ^k distinct words of length k, but a word y of length n has exactly $n + 1 - k$ factors of length k. Thus, if $\sigma^k > n + 1 - k$ there are absent words of size k in y. Consequently we have $\ell \leq \log_\sigma(n + 1 - \ell) < \log_\sigma(n)$. Thus, we compute ℓ, the length of the shortest minimal absent word, in one pass over the LCP array by counting the number of positions having an LCP value equal to d, for all $d \in [0, \lfloor \log_\sigma(n) \rfloor]$.

According to Lemma 2 we can ignore positions having an LCP value lower than $\ell - 2$ when computing minimal absent words. Hence, we focus on LCP-intervals of LCP-depth above or equal to $\ell - 2$: they are sufficient to exhaustively compute the set of minimal absent words. Consequently we compute

(a) Elapsed-time comparison (b) Relative speed-up of pMAW

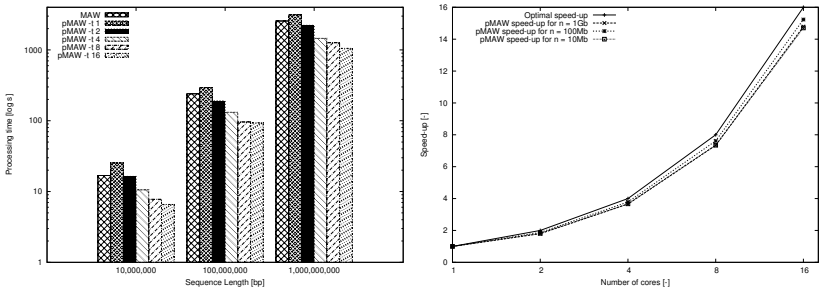


Fig. 3: Elapsed-time comparison of pMAW and MAW and relative speed-up of pMAW for computing minimal absent words using synthetic DNA sequences

the set of positions k_i with i in $[0, (\sigma-1)\sigma^{\ell-3}]$ such that $\mathsf{LCP}[k_i] = \ell - 3$. $[0, k_0), [k_0, k_1), \ldots, [k_{m-1}, k_m), [k_m, n)$, with $m = (\sigma-1)\sigma^{\ell-3}$, is a partition of $[0, n-1]$. This partition is such that, every LCP-interval of LCP-depth above or equal to $\ell - 2$ is entirely included in one of the sub-intervals $[k_i, k_{i+1})$.

Therefore we can consider each one of these sub-intervals *independently*, and thus parallelise the computation of minimal absent words. In each sub-interval we go through the SA and LCP arrays starting at the first (from left to right) local maximum and going down until we reach a local minimum, as described in Section 3.2. For an overview of the algorithm pMAW inspect Fig. 2.

4 Experimental Results

We implemented algorithm pMAW as a programme to compute all minimal absent words of a given sequence. The programme was implemented in the C programming language, using Open Multi-Processing (OpenMP) API for shared-memory multiprocessing programming, and developed under GNU/Linux operating system. It takes as input arguments a file in (Multi)FASTA format and the minimal and maximal length of minimal absent words to be outputted; and then produces a file with all minimal absent words of length within this range as output. There are additional input parameters; for example, the number t of available processing elements. The implementation is distributed under the GNU General Public License (GPL), and it is available at `http://github.com/solonas13/maw`, which is set up for maintaining the source code and the man-page documentation. The experiments were conducted on a Desktop PC using 1 to 16 cores of 2 Intel Xeon E5-2670V2 Ten-Core CPUs at 2.50GHz and 256GB of main memory under 64-bit GNU/Linux.

To evaluate the efficiency of our implementation, we compared it against the corresponding performance of MAW [3], which is currently the fastest available implementation for computing minimal absent words. We generated three ran-

(a) Elapsed-time comparison (b) Relative speed-up of pMAW

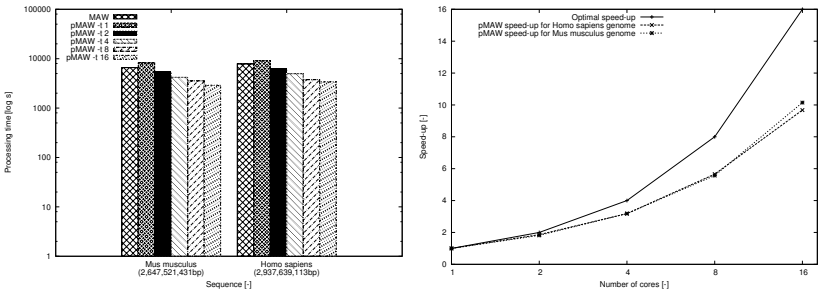


Fig. 4: Elapsed-time comparison of pMAW and MAW and relative speed-up of pMAW for computing minimal absent words using real DNA sequences

dom sequences of length 10Mbp, 100Mbp, and 1Gbp, respectively, by using a uniform frequency distribution of letters of the DNA alphabet. We computed all minimal absent words of length at most 20 for each sequence. We considered both the $5' \rightarrow 3'$ and the $3' \rightarrow 5'$ DNA strands. Fig. 3a depicts elapsed-time comparisons of pMAW and MAW, *including* the sequential part of the algorithm. pMAW becomes the fastest in *all* cases when $t \geq 2$ accelerating the computation by more than a factor of two when $t = 16$. Notice that the y-axis is on logarithmic scale. The measured relative speed-up of pMAW is illustrated in Fig. 3b. The relative speed-up was calculated as the ratio of the runtime of pMAW on 1 core to the runtime of pMAW on t cores, excluding the sequential part of the algorithm. The results highlight the *excellent* scalability of pMAW when the letters have a uniform frequency distribution in the sequence. In this case, pMAW achieves near-optimal speed-ups, confirming our theoretical findings.

To further evaluate the efficiency of our implementation, we compared it against the corresponding performance of MAW using real data. We considered the genomes of *Homo sapiens* and *Mus musculus*, obtained from the NCBI database (`ftp://ftp.ncbi.nih.gov/genomes/`). We computed all minimal absent words of length at most 20 of the complete sequence of the *Homo sapiens* $(2,937,639,113\text{bp})$ and *Mus musculus* $(2,647,521,431\text{bp})$ genomes—ignoring unknown bases. We considered both the $5' \rightarrow 3'$ and the $3' \rightarrow 5'$ DNA strands. Fig. 4a depicts elapsed-time comparisons of pMAW and MAW, *including* the sequential part of the algorithm. pMAW becomes the fastest in *all* cases when $t \geq 2$ accelerating the computation by more than a factor of two when $t = 16$. Notice that the y-axis is on logarithmic scale. The measured relative speed-up of pMAW is illustrated in Fig. 4b. The relative speed-up was calculated as the ratio of the runtime of pMAW on 1 core to the runtime of pMAW on t cores, excluding the sequential part of the algorithm. The results highlight the *good* scalability of pMAW with real data. The computation is accelerated by a factor of 10 when $t = 16$. The maximum allocated memory was 137GB for both programmes.

5 Final Remarks

The importance of our contribution here is underlined by the fact that any parallel algorithms for the construction of the involved indexing data structure can be used *directly* to replace the sequential part of the algorithm proposed here (see Fig. 2). This would result in a *fully* parallel algorithm for the computation of minimal absent words. Our immediate target is to investigate the performance of such an algorithm by using the parallel algorithms presented in [16] for constructing the suffix array and the longest common prefix array.

References

1. Abboud, A., Williams, V.V., Weimann, O.: Consequences of faster alignment of sequences. In: 41st ICALP, Part I. pp. 39–51. LNCS, Springer (2014)
2. Acquisti, C., Poste, G., Curtiss, D., Kumar, S.: Nullomers: Really a matter of natural selection? PLoS ONE 2(10) (2007)
3. Barton, C., Heliou, A., Mouchard, L., Pissis, S.P.: Linear-time computation of minimal absent words using suffix array. BMC Bioinformatics 15, 388 (2014)
4. Belazzougui, D., Cunial, F., Kärkkäinen, J., Mäkinen, V.: Versatile succinct representations of the bidirectional Burrows-Wheeler transform. In: 21st ESA. pp. 133–144. LNCS, Springer (2013)
5. Chairungsee, S., Crochemore, M.: Using minimal absent words to build phylogeny. Theoretical Computer Science 450(0), 109–116 (2012)
6. Crochemore, M., Mignosi, F., Restivo, A.: Automata and forbidden words. Information Processing Letters 67, 111–117 (1998)
7. Fischer, J.: Inducing the LCP-Array. In: Dehne, F., Iacono, J., Sack, J.R. (eds.) 12th WADS. LNCS, vol. 6844, pp. 374–385. Springer (2011)
8. Garcia, S.P., Pinho, A.J.: Minimal Absent Words in Four Human Genome Assemblies. PLoS ONE 6(12) (2011)
9. Garcia, S.P., Pinho, O.J., Rodrigues, J.M.O.S., Bastos, C.A.C., G, P.J.S.: Minimal absent words in prokaryotic and eukaryotic genomes. PLoS ONE 6 (2011)
10. Haubold, B., Pierstorff, N., Möller, F., Wiehe, T.: Genome comparison without alignment using shortest unique substrings. BMC Bioinformatics 6, 123 (2005)
11. Jacobson, G.: Space-efficient static trees and graphs. In: 30th SFCS. pp. 549–554. SFCS '89, IEEE Computer Society (1989)
12. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. 22(5), 935–948 (1993)
13. Mignosi, F., Restivo, A., Sciortino, M.: Words and forbidden factors. Theoretical Computer Science 273(1-2), 99–117 (2002)
14. Nong, G., Zhang, S., Chan, W.H.: Linear suffix array construction by almost pure induced-sorting. In: DCC 2009. pp. 193–202. IEEE Computer Society (2009)
15. Pinho, A.J., Ferreira, P.J.S.G., Garcia, S.P.: On finding minimal absent words. BMC Bioinformatics 11 (2009)
16. Shun, J.: Fast parallel computation of longest common prefixes. In: SC 2014. pp. 387–398. IEEE Computer Society (2014)
17. Silva, R.M., Pratas, D., Castro, L., Pinho, A.J., Ferreira, P.J.S.G.: Three minimal sequences found in ebola virus genomes and absent from human DNA. Bioinformatics (2015)
18. Wu, Z.D., Jiang, T., Su, W.J.: Efficient computation of shortest absent words in a genomic sequence. Information Processing Letters 110(14-15), 596 – 601 (2010)