Towards optimal packed string matching

Ben-Kiki, Oren; Bille, Philip; Breslauer, Dany; Gasieniec, Leszek; Grossi, Roberto; Weimann, Oren

Published in:
Theoretical Computer Science

Link to article, DOI:
10.1016/j.tcs.2013.06.013

Publication date:
2014

Document Version
Peer reviewed version

Towards Optimal Packed String Matching[1]

Oren Ben-Kiki[a], Philip Bille[b], Dany Breslauer[c], Leszek Gąsieniec[d], Roberto Grossi[e], Oren Weimann[f]

[a]*Intel Research and Development Center, Haifa, Israel.*
[b]*Technical University of Denmark, DTU Compute, Copenhagen, Denmark.*
[c]*Caesarea Rothschild Institute, University of Haifa, Haifa, Israel.* [2]
[d]*University of Liverpool, Liverpool, United Kingdom.*
[e]*Dipartimento di Informatica, Università di Pisa, Pisa, Italy.* [3]
[f]*Computer Science Department, University of Haifa, Haifa, Israel.*

Abstract

In the packed string matching problem, it is assumed that each machine word can accommodate up to α characters, thus an n-character string occupies n/α memory words.

(a) We extend the Crochemore-Perrin constant-space $O(n)$-time string matching algorithm to run in optimal $O(n/\alpha)$ time and even in real-time, achieving a factor α speedup over traditional algorithms that examine each character individually. Our macro-level algorithm only uses the standard AC^0 instructions of the word-RAM model (i.e. no integer multiplication) plus two specialized micro-level AC^0 word-size packed-string instructions. The main *word-size string-matching* instruction WSSM is available in contemporary commodity processors. The other *word-size maximum-suffix* instruction WSLM is only required during the pattern pre-processing. Benchmarks show that our solution can be efficiently implemented, unlike some prior theoretical packed string matching work.

(b) We also consider the complexity of the packed string matching problem in the classical word-RAM model in the absence of the specialized micro-level instructions WSSM and WSLM. We propose micro-level algorithms for the theoretically efficient emulation using parallel algorithms techniques to emulate WSSM and using the Four-Russians technique to emulate WSLM. Surprisingly, our bit-parallel emulation of WSSM also leads to a new simplified parallel random access machine string matching algorithm. As a byproduct to facilitate our results we develop a new algorithm for finding the leftmost (most significant) `1` bits in consecutive non-overlapping blocks of uniform size inside a word. This latter problem is not known to be reducible to finding the rightmost `1`, which can be easily solved, since we do not know how to reverse the bits of a word in $O(1)$ time.

Keywords: String matching, word-RAM, packed strings.

[1] Preliminary versions of this work were presented at FSTTCS 2011 [9] and at CPM 2012 [16].

[2] Partially supported by the European Research Council (ERC) Project SFEROT and by the Israeli Science Foundation Grants 686/07, 347/09 and 864/11.

[3] Partially supported by Italian project PRIN AlgoDEEP (2008TFBWL4) of MIUR.

Dedicated to Professor Gad M. Landau, on the occasion of his 60^{th} *birthday.*

1. Introduction

Hundreds of articles, literally, have been published about string matching, exploring the multitude of theoretical and practical facets of this fascinating fundamental problem. For an n-character text T and an m-character pattern x, the classical algorithm by Knuth, Morris and Pratt [47] takes $O(n+m)$ time and uses $O(m)$ auxiliary space to find all pattern occurrences in the text, namely, all text positions i, such that $x = T[i \ldots i+m-1]$. Many other algorithms have been published; some are faster on the average, use only constant auxiliary space, operate in real-time, or have other interesting benefits. In an extensive study, Faro and Lecroq [30, 31] offer an experimental comparative evaluation of some 85 string matching algorithms.

Packed strings. In modern computers, the size of a machine word is typically larger than the size of an alphabet character and the machine level instructions may operate on whole words, i.e., 64-bit or longer words vs. 8-bit ASCII, 16-bit UCS, 2-bit 4-character biological DNA, 5-bit 20-character amino acid alphabets, etc. The *packed string representation* fits multiple characters into one larger word, so that the characters can be compared in bulk rather than individually: if the characters of a string are drawn from an alphabet Σ, then a word of $\omega \geq \log_2 n$ bits fits up to α characters, where the packing factor is $\alpha = \frac{\omega}{\log_2 |\Sigma|} \geq \log_{|\Sigma|} n$. Throughout the paper, we assume that $|\Sigma|$ is a power of two, ω is divisible by $\log_2 |\Sigma|$, and the packing factor α is a whole integer. Observe that larger words must come at some cost in reality, but that computer hardware is highly optimized to efficiently realize some standard instruction set, leading to our working assumption that the available instructions that operate on ω-bit words take constant time.

Using packed representation for string matching is not a new idea and goes back to early string matching papers by Knuth, Morris and Pratt [47, §4] and Boyer and Moore [12, §8-9], to times when hardware byte addressing was new and often even less efficient than word addressing. Since then, several practical solutions that take advantage of the packed string representation have been proposed in the literature, e.g. [7, 10, 29, 35, 36, 53, 55]. However, none of these practical algorithms improves over the theoretical worst-case $O(n)$ time bounds of the traditional algorithms. On the other hand, any string matching algorithm should take at least $\Omega(n/\alpha)$ time to read a packed text in the worst case, so there remained a gap to fill. Note that on the average, it is not even required to examine all the text characters [12, 22, 53, 58] and in fact, those algorithms that manage to skip parts of the text are often the fastest in practice.

Model of computation. We adopt the word-RAM model with ω-bit words where we restrict to only AC^0 instructions (i.e., arithmetic, bit-wise, and shift operations but no integer multiplication) plus two other specialized AC^0 instructions described below. Let $[d]$ denote the set $\{0, 1, \ldots, d-1\}$.

- **Word-Size String Matching** (WSSM): find the occurrences of one short pattern x that fits in one word (up to α characters) in a text y that fits in two words (up to $2\alpha - 1$ characters). The output is a binary word Z of $2\alpha - 1$ bits such that its i-th bit $Z[i] = \mathtt{1}$ if and only if $y[i \dots i + |x| - 1] = x$. When $i + |x| - 1 \geq 2\alpha - 1$, this means that only a prefix of x is matched.

  Remark: WSSM is available in the *Advanced String Operations* in *Streaming SIMD Extension (SSE4.2) and in Advanced Vector Extension (AVX) Efficient Accelerated String and Text Processing* instruction sets in contemporary commodity Intel and AMD processors [3, 44, 46].

- **Word-Size Lexicographically Maximum Suffix** (WSLM): given a packed string x that fits in one word (up to α characters), return position $i \in [\alpha]$ such that the suffix $x[i \dots \alpha-1]$ is lexicographically maximum among the suffixes in $\{x[j \dots \alpha-1] \mid j \in [\alpha]\}$.

  Remark: WSLM is only used in the pattern pre-processing.

Existing work. A significant theoretical step recently taken introduces a few solutions based on either tabulation (a.k.a. "the Four-Russians trick") or word-level parallelism (a.k.a. "bit-parallelism"). A summary of the known bounds and our new results is given in Table 1.

Time	Space	Reference
$O(\frac{n}{\log_{\lvert\Sigma\rvert} n} + n^\varepsilon m + occ)$	$O(n^\varepsilon m)$	Fredriksson [35, 36]
$O(\frac{n}{\log_{\lvert\Sigma\rvert} n} + m + occ)$	$O(n^\varepsilon + m)$	Bille [11]
$O(\frac{n}{\alpha} + \frac{n}{m} + m + occ)$	$O(m)$	Belazzougui [8]
$O(\frac{n}{\alpha} + \frac{m}{\alpha} + occ)$	$O(1)$	This paper, using WSSM and WSLM

Table 1: Comparison of packed-string matching algorithms.

Our new result uses the two specialized packed-string instructions WSSM and WSLM. Fredriksson [35, 36] used tabulation and obtained an algorithm that uses $O(n^\varepsilon m)$ space and $O(\frac{n}{\log_{|\Sigma|} n} + n^\varepsilon m + occ)$ time, where occ denotes the number of pattern occurrences and $\varepsilon > 0$ denotes an arbitrary small constant. Bille [11] improved these bounds to $O(n^\varepsilon + m)$ space and $O(\frac{n}{\log_{|\Sigma|} n} + m + occ)$ time. Very recently, Belazzougui [8] showed how to use word-level parallelism to obtain $O(m)$ space and $O(\frac{n}{m} + \frac{n}{\alpha} + m + occ)$ time. Belazzougui's algorithm uses a number of succinct data structures as well as hashing techniques. For $\alpha \leq m \leq n/\alpha$, his time complexity is optimal while space complexity is not. As reported by the above authors, none of these theoretical results is practical.

Our result. We propose an $O(n/\alpha + m/\alpha)$ time string matching algorithm (where the term m/α is kept for comparison with the other results) that is derived from the elegant Crochemore-Perrin [24] algorithm. The Crochemore-Perrin algorithm takes linear time, uses only constant auxiliary space, and can be implemented in real-time following the recent work by Breslauer, Grossi and Mignosi [17] – benefits that are also enjoyed in our packed string settings. The algorithm has an attractive property that it compares the text characters only moving forward on two wavefronts without ever having to back up, relying on the celebrated Critical Factorization Theorem [19, 49].

We use WSSM to anchor the pattern in the text and continue with bulk character comparisons that match the remainder of the pattern. Our reliance on a specialized packed-string matching instruction is not far fetched, given the recent availability of such instructions in commodity processors, which has been a catalyst for our work. The output occurrences are identified compactly in a bit mask that can be spelled out as an extensive list of text positions in extra $O(occ)$ time and $O(1)$ space.

Unlike prior theoretical work, our solution has a cache-friendly sequential memory access without using large external tables or succinct data structures, and therefore, can also be efficiently implemented. The same specialized packed string matching instruction could also be used in other string matching and string processing algorithms, e.g. the Knuth-Morris-Pratt algorithm [45, §10.3.3], but our algorithm has the additional advantages that it also works in real-time and uses only constant auxiliary space. We hope that algorithm design using non-standard specialized instructions available in commodity processors is going to continue and evolve, where the variety of available specialized instructions and the algorithmic design work will increasingly cross-fertilize.

Specialized instruction emulation. If the two specialized packed string instructions WSSM and WSLM are not available, then we can emulate these instructions. Our proposed emulations cause a small slowdown as shown in Table 2.

	Time	*Space*	*Emulation*
Text	$O(\frac{n}{\alpha}\log\omega + occ)$	$O(1)$	bit-parallel WSSM no pre-processing
	$O(\frac{n}{\alpha} + \alpha + occ)$	$O(1)$	bit-parallel WSSM pre-processing
Pattern	$O(m)$	$O(1)$	sequential WSLM no pre-processing
	$O(\frac{m}{\log_{\lvert\Sigma\rvert} n})$	$O(n^{\epsilon})$	bit-parallel WSLM pre-processing (Four-Russians trick)

Table 2: Packed string matching using the standard word-RAM emulation of ω-bit α-character WSSM and WSLM instructions.

While the classical Four-Russians trick [6] can be used to emulate either of the two specialized instructions, its lookup table space requirement often makes it impractical and typically limits the packing factor to $\alpha = \Theta(\log_{|\Sigma|} n)$. For the WSSM instruction, we offer a better constant time bit-parallel emulations using the standard word-RAM instructions, including integer multiplication, that is built upon techniques that were developed for the *Parallel Random Access Machine* model [13, 14, 15, 20, 23, 26, 38, 39, 40, 41, 56, 57]: surprisingly, it also leads to a greatly simplified algorithm in that model. This bit-parallel emulation requires a slower pattern pre-processing. An alternative emulation does not require a special pattern pre-processing, but requires $\log\omega$-bit padding resulting in larger $\omega\log\omega$-bit words or ω-bit words and a $\log\omega$ slowdown.

Finally, we mention that our bit-parallel emulation of WSSM also leads to a new $O(1)$-time algorithm for finding the leftmost (most significant) `1` bits in consecutive non-overlapping blocks of uniform size inside a word of ω bits. This problem has a clear resemblance with finding the rightmost `1` inside each block, solved in [32, 48], but we do not know how to reverse the bits of a word in $O(1)$ time. In fact, bit reversal is known to require $\Omega(\log \omega)$ time without the use of integer multiplication [18].

Paper organization. We start by briefly reviewing simple bit-wise operations in Section 2. In Section 3 we describe the reduction of the packed string matching problem to the two specialized packed string instructions WSSM and WSLM, using the Crochemore-Perrin algorithm.

We then move on to the standard word-RAM model (without WSSM and WSLM): In Section 4 we show how to emulate WSSM and WSLM in the standard word-RAM model, assuming a constant time operation that finds the leftmost `1` bits in consecutive blocks inside a word. In Section 5 we present this constant time operation.

We report on some experimental results with the WSSM instruction on contemporary processors in Section 6, and finish with conclusions and open problems in Section 7.

2. Bit-wise Operations

This work is largely based on our ability to compare multiple characters of a packed string in an efficient single machine level instruction that operates on large machine words, exploiting the bit level parallelism inherent in the standard RAM bit-wise and arithmetic instructions. Knuth [48, §7.1.3] provides a comprehensive introduction to bit-wise tricks and techniques, while Fich [32] describes some of the operations that are used in this paper. We briefly introduce and review the notations and operations that we use.

Recall that the class AC^0 consist of problems that admit polynomial size circuits of depth $O(1)$, with Boolean `and`/`or` gates of unbounded fan-in and `not` gates only at the inputs. In the circuit complexity sense, all standard arithmetic, bit-wise, and shift operations are AC^0, with the exception of integer multiplication (division) which is not AC^0 since it can be used to compute parity [37]. The specialized instructions, WSSM and WSLM, are both AC^0. However, some of the primitives that we use to compute them, including computing the leftmost `1`-bits in consecutive non-overlapping blocks of uniform size in constant time, do not easily map to standard word-RAM AC^0 operations. Recall that computing the leftmost `1` by reversing is known to require $\Omega(\log \omega)$ time without the use of integer multiplication [18][4]. As we shall see, integer multiplication turns to be very useful in realizing such primitives in the standard word-RAM model.

[4] Fich [32] gives an $O(1)$ procedure for computing the leftmost `1` but her solution requires a floating-point representation.

instruction	*meaning* (operations are modulo 2^ω)	*reference*
$X = \mathtt{and}(A, B)$	*bit AND*, where $X_i = (A_i \wedge B_i)$, for all $i = 0, \ldots, \omega - 1$	RAM
$X = \mathtt{or}(A, B)$	*bit OR*, where $X_i = (A_i \vee B_i)$, for all $i = 0, \ldots, \omega - 1$	RAM
$X = \mathtt{neg}(A)$	*bit negation*, where $X_i = \neg A_i$, for all $i = 0, \ldots, \omega - 1$	RAM
$X = \mathtt{xor}(A, B)$	*bit exclusive OR*, where $X_i = (A_i \vee B_i)$, for all $i = 0, \ldots, \omega - 1$	RAM
$X = \mathtt{add}(A, B)$	*addition*, where X satisfies $V(X) = V(A) + V(B)$	RAM
$X = \mathtt{sub}(A, B)$	*subtraction*, where X satisfies $V(X) = V(A) - V(B)$	RAM
$X = \mathtt{mul}(A, B)$	*multiplication*, where X satisfies $V(X) = V(A) \cdot V(B)$	RAM
$X = \mathtt{shl}(A, k)$	*shift left* by k positions, where X satisfies $V(X) = V(A) \cdot 2^k$	RAM
$X = \mathtt{shr}(A, k)$	*shift right* by k positions, where X satisfies $V(X) = V(A)/2^k$	RAM
$X = \mathtt{rmo}(A, k)$	rightmost `1` in each consecutive block of size k i.e., $(X_i = \mathtt{1})$ if and only if $(A_i = \mathtt{1}) \wedge (A_{i-l} = \mathtt{0})$ for all $0 \leq l < (i \bmod k)$	[32, 48]
$X = \mathtt{lmo}(A, k)$	leftmost `1` in each consecutive block of size k i.e., $(X_i = \mathtt{1})$ if and only if $(A_i = \mathtt{1}) \wedge (A_{i+l} = \mathtt{0})$ for all $(i \bmod k) < l < k$	Sect. 5

Table 3: Bit-wise operations required by our algorithms in the RAM model.

2.1. Instruction set

Consider the set of binary words $\mathcal{B} = \{\mathtt{0}, \mathtt{1}\}^\omega$. In our algorithms, we use a number of constant time operations defined on whole words from $\mathcal{B}$ or their partitions into consecutive blocks of uniform length. We assume also that indices of symbols in words of $\mathcal{B}$ are enumerated from $w - 1$ down to 0, counting from the left. This notation is adopted to reflect a natural representation of polynomials. To be precise, any word $A \in \mathcal{B}$ can be interpreted as the polynomial $P_A(x) = A_{\omega-1} \cdot x^{\omega-1} + A_{\omega-2} \cdot x^{\omega-2} + \cdots + A_1 \cdot x^1 + A_0 \cdot x^0$, where $V(A)$ is defined as $P_A(2)$. For this reason we will also refer to elements of $\mathcal{B}$ as *vectors*.

Table 3 shows the operations on words from $\mathcal{B}$ that will be of our interest: they are standard in the RAM model, except the last two. It has been shown in [32, 48] that `rmo` operation can be implemented in $O(1)$ time in the model adopted in this paper. In contrast, a fast implementation of `lmo` operation is not known, and the main contribution of Section 5 is to show how to implement this operation in $O(1)$ time.

We will also use the following lemma, which can be derived from [32]. Namely, it follows from the fact that (1) concatenating L copies of the same string of length k can be done in $O(1)$ time by simply multiplying the string by $1(0^{k-1}1)^{L-1}$, and (2) concatenating two strings x, y can be done in $O(1)$ time by shifting-left x by $|y|$ positions and then doing OR with y.

Lemma 2.1. *Any word from $\mathcal{B}$ of the form $(\mathtt{0}^a(\mathtt{0}^b\mathtt{1}^c\mathtt{0}^d)^e\mathtt{0}^f)$, for non-negative integer values a, b, c, d, e, f such that $a + f + e(b + c + d) = \omega$, can be generated in $O(1)$ time.*

3. Packed String Matching: Macro-Level Algorithms

We now describe how to solve the *packed string matching* problem using the two specialized word-size string matching instructions WSSM and WSLM, the bit-parallel operations in Section 2.1, and the standard word-RAM bulk comparisons of packed strings.

Theorem 3.1. *Packed string matching for a length-m pattern and a length-n text can be solved in $O(\frac{m}{\alpha} + \frac{n}{\alpha})$ time in the word-RAM extended with constant time* WSSM *and* WSLM *instructions. Listing explicitly the occ text positions of the pattern occurrences takes an additional $O(occ)$ time. The algorithm can be made real-time, using just $O(1)$ auxiliary words of memory besides the read-only $\frac{m}{\alpha} + \frac{n}{\alpha}$ words that store the inputs.*

The algorithm behind Theorem 3.1 follows the classical scheme, in which a text scanning phase is run after the pattern pre-processing. In the following, we first present the necessary background and then describe how to perform the text scanning phase using WSSM, and the pattern pre-processing using WSLM.

3.1. Background

Properties of periodic strings are often used in many efficient string algorithms, e.g. [12, 17, 19, 25, 24, 33, 42, 47, 49]. We assume that the reader is familiar with the Crochemore-Perrin algorithm [24] and its real-time variation Breslauer-Grossi-Mignosi [17] and briefly review these algorithms and introduce their adaptation to packed strings.

Period. A string u is *a period* of a string x if x is a prefix of u^k for some integer k, or equivalently if x is a prefix of ux. The shortest period of x is called *the period* of x and its length is denoted by $\pi(x)$. Observe that close-by overlapping occurrences of the pattern imply that there is a self-overlap of the pattern, or in other words, a period.

Critical Factorization. A *substring* or a *factor* of a string x is a contiguous block of symbols u, such that $x = x'ux''$ for two strings x' and x''. A *factorization* of x is a way to break x into a number of factors. We consider factorizations of a string $x = uv$ into two factors: a *prefix* u and a *suffix* v. Such a factorization can be represented by a single integer and is *non-trivial* if neither of the two factors is equal to the empty string.

Given a factorization $x = uv$, a *local period* of the factorization is defined as a non-empty string p that is consistent with both sides u and v. Namely, (i) p is a suffix of u or u is a suffix of p, and (ii) p is a prefix of v or v is a prefix of p. The shortest local period of a factorization is called *the local period* and its length is denoted by $\mu(u, v)$. A non-trivial factorization $x = uv$ is called a *critical factorization* if the local period of the factorization is of the same length as the period of x, i.e., $\mu(u, v) = \pi(uv)$. See Figure 1.

```
  a | b a a a b a           a b | a a a b a         a b a | a a b a
b a   b a               a a a b   a a a b               a   a
     (a)                         (b)                       (c)
```

Figure 1: The local periods at the first three non-trivial factorizations of the string abaaaba. In some cases the local period overflows on either side; this happens when the local period is longer than either of the two factors. The factorization (b) is a critical factorization with local period aaab of the same length as the global period abaa.

Theorem 3.2. *(Critical Factorization Theorem, Cesari and Vincent [19, 49]) Given any $\pi(x) - 1$ consecutive non-trivial factorizations of a string x, at least one is critical.*

The Crochemore-Perrin algorithm. While other algorithms may be used with the WSSM instruction, the Crochemore-Perrin algorithm is particularly attractive because of its simple text processing. Recall that Crochemore and Perrin use Theorem 3.2 to break up the pattern as critical factorization $x = uv$ with non-empty prefix u and suffix v, such that $|u| \leq \pi(x)$. Then, they exploit the critical factorization of $x = uv$ by matching the longest prefix z of v against the current text symbols, and using Theorem 3.3 whenever a mismatch is found.

Theorem 3.3. *(Crochemore and Perrin [24]) Let $x = uv$ be a critical factorization of the pattern and let p be any local period at this factorization, such that $|p| \leq \max(|u|, |v|)$. Then $|p|$ is a multiple of $\pi(x)$, the period length of the pattern.*

Precisely, if $z = v$, they show how to declare an occurrence of x. Otherwise, the symbol following z in v is mismatching when compared to the corresponding text symbol, and the pattern x can be *safely* shifted by $|z| + 1$ positions to the right (there are other issues for which we refer the reader to [24]).

To simplify the matter in the rest of the section, we discuss how to match the pattern suffix v assuming without loss of generality that $|u| \leq |v|$. Indeed, if $|u| > |v|$, the Crochemore-Perrin approach can be simplified as shown in [17]: use two critical factorizations, $x = uv$ and $x' = u'v'$, for a prefix x' of x such that $|x'| > |u|$ and $|u'| \leq |v'|$. In this way, matching both u' and v' suitably displaced by $|x| - |x'|$ positions from matching v, guarantees that x occurs. This fact enables us to focus on matching v and v', since the cost of matching u' is always dominated by the cost of matching v', and we do not need to match u. For the sake of discussion, it suffices to consider only one instance, namely, suffix v.

We now give more details on the text processing phase, assuming that the pattern preprocessing phase has correctly found the critical factorization of the pattern x and its period $\pi(x)$, and any additional pattern pre-processing that may be required (Section 3.3).

3.2. Packed text processing

The text processing has complementary parts that handle short patterns and long patterns. A pattern x is *short* if its length is at most α, namely, the packed pattern fits into a single word, and is *long* otherwise. Processing short patterns is immediate with WSSM and, as we shall see, the search for long patterns reduces to that for short patterns.

Short patterns. When the pattern is already short, WSSM is repeatedly used to directly find all occurrences of the pattern in the text.

Lemma 3.4. *There exists an algorithm that finds all occurrences of a short pattern of length $m \leq \alpha$ in a text of length n in $O\left(\frac{n}{\alpha}\right)$ time using $O(1)$ auxiliary space.*

Proof. Consider the packed text blocks of length $\alpha + m - 1$ that start on word boundaries, where each block overlaps the last $m - 1$ characters of the previous block and the last block might be shorter. Each occurrence of the pattern in the text is contained in exactly one such block. Repeatedly use the WSSM instruction to search for the pattern of length $m \leq \alpha$ in these text blocks whose length is at most $\alpha + m - 1 \leq 2\alpha - 1$. □

Long patterns. Let x be a long pattern of length $m > \alpha$: occurrences of the pattern in the text must always be spaced at least the period $\pi(x)$ locations apart. We first consider the easier case where the pattern has a long period, namely $m \geq \pi(x) \geq \alpha$, and so there is at most one occurrence starting within each word.

Lemma 3.5. *There exists an algorithm that finds all occurrences of a long-period long-pattern x of length $m \geq \pi(x) \geq \alpha$, in a text of length n in $O\big(\frac{n}{\alpha}\big)$ time using $O(1)$ auxiliary space.*

Proof. The Crochemore-Perrin algorithm can be naturally implemented using the WSSM instruction and bulk character comparisons. Given the critical factorization $x = uv$, the algorithm repeatedly searches using WSSM for an occurrence of a prefix of v of length $\min(|v|, \alpha)$ starting in each packed word aligned with v, until such an occurrence is discovered. If more than one occurrence is found starting within the same word, then by Theorem 3.3, only the first such occurrence is of interest.

The algorithm then uses the occurrence of the prefix of v to anchor the pattern within the text and continues to compare the rest of v with the aligned text and then compares the pattern prefix u, both using bulk comparison of words containing α packed characters. Bulk comparisons are done by comparing words (comparing words A and B is done by checking if $\texttt{xor}(A, B) = \texttt{0}$). In case of a mismatch, the mismatch position can be found by the most significant `1` bit via $\texttt{lmo}(\texttt{xor}(A, B))$.

A mismatch during the attempt to verify the suffix v allows the algorithm to shift the pattern ahead until v is aligned with the text after the mismatch. A mismatch during the attempt to verify u, or after successfully matching u, causes the algorithm to shift the pattern ahead by $\pi(x)$ locations. In either case the time adds up to only $O\big(\frac{n}{\alpha}\big)$. $\square$

When the period of the pattern is shorter than the word size, that is $m > \alpha > \pi(x)$, there may be several occurrences of the pattern starting within each word. The algorithm is very similar to the long period algorithm above, but with special care to efficiently manipulate the bit masks representing all the occurrences.

Lemma 3.6. *There exists an algorithm that finds all occurrences of a short-period long-pattern x of length $m > \alpha > \pi(x)$, in a text of length n in $O\big(\frac{n}{\alpha}\big)$ time using $O(1)$ auxiliary space.*

Proof. Let p be the period of x of length $\pi(x)$, and write $x = p^r p'$, where p' is a prefix of p. If we can find the maximal runs of consecutive occurrences of p inside the text, then it is easy to locate the occurrences of x. To this end, let $k \leq r$ be the maximum positive integer such that $|p^k| = k \cdot \pi(x) \leq \alpha$. Note that there cannot exist two occurrences of p^k that are completely inside the same word.

We examine one word w of the text at a time while maintaining the current run of consecutive occurrences of p spanning the text word w' preceding w. More formally, while scanning two consecutive words $w'w$ of the text we maintain three auxiliary words: $e(w')$

stores the rightmost index in w' that *ends* a maximal run of consecutive occurrences of p starting somewhere in the text before w' and ending in w'; $\ell(w')$ stores the *length* of this maximal run (that ends in $e(w')$); $s(w')$ stores the rightmost index in w' that *starts* an occurrence of p^k whose matching substring is completely inside $w'w$.

Given $e(w')$ and $\ell(w')$, we show how to compute $e(w)$ and $\ell(w)$. We first compute $s(w')$ by applying WSSM to p^k and $w'w$ and taking the rightmost occurrence completely inside $w'w$ that starts in w'. To do so, recall that the output of WSSM is a binary word Z of $2\alpha - 1$ bits such that $Z[i] = \mathtt{1}$ if and only if an occurrence of p^k starts in index i. To avoid matching only prefixes of p^k we set to $\mathtt{0}$ all bits i in Z such that $i \geq 2\alpha - |p^k|$. This is easily achieved by doing $Z' = \mathtt{and}(Z, M)$ with the bit mask $M = \mathtt{1}^{|p^k|-1}\mathtt{0}^{2\alpha-|p^k|}$ built using Lemma 2.1. Finally, we compute $s(w')$ by using $\mathtt{rmo}$ on Z'.

Once we have $s(w')$ we move on to computing $e(w)$ and $\ell(w)$. If $s(w') = e(w') + 1$ (i.e., the previous maximal run of occurrences of p is aligned with the current run of occurrences of p so we can extend it) we set $e(w) = e(w') + |p^k| - \alpha$ and $\ell(w) = \ell(w') + |p^k|$. At this point we also check if $\ell(w) \geq |p^r p'|$, if so we have identified occurrences of $x = p^r p'$ and we can report all of them by simple arithmetics. If on the other hand $s(w') \neq e(w') + 1$ (there is no occurrence of p^k, or there is but it is not aligned with the current run of occurrences of p), we set $e(w) = s(w') + |p^k| - 1 - \alpha$ and $\ell(w) = |p^k|$. At this point, we use WSSM to find p' in $w'w$. We then check if p' occurs right after $e(w')$. If so, and also $\ell(w') \geq |p^r|$ then we have also identified occurrences of $x = p^r p'$. □

Real-time algorithm. As mentioned in Section 3.1, the Crochemore-Perrin algorithm can be implemented in real time using two instances of the basic algorithm with carefully chosen critical factorizations [17]. Since we are following the same scheme here, with straightforward proper care to align the resulting output bit masks from the two instances, our algorithm reports the output bit mask of the full pattern occurrences ending in each text word in $O(1)$ time after reading the word. Thus, we can obtain a real-time version of the packed string matching algorithm as claimed in Theorem 3.1.

3.3. Packed pattern pre-processing

Given the pattern x, the pattern pre-processing of Crochemore-Perrin produces the period length $\pi(x)$ and a critical factorization $x = uv$ (Section 3.1): for the latter, they show that v is the lexicographically maximum suffix in the pattern under either the regular alphabet order or its inverse order, and use the algorithm by Duval [28]. The pattern pre-processing of Breslauer, Grossi and Mignosi [17] uses Crochemore-Perrin pre-processing, and it also requires the prefix x' of x such that $|x'| > |u|$ and its critical factorization $x' = u'v'$ where $|u'| \leq |v'|$. We thus end up with only the following two problems:

1. Given a string x, find its lexicographically maximum suffix v (under the regular alphabet order or its inverse order).
2. Given a string $x = uv$, find its period $\pi(x)$.

When $m = O(\frac{n}{\alpha})$, which is probably the case in many situations, we can simply run the above algorithms in $O(m)$ time to solve the above two problems. We focus here on the case when $m = \Omega(\frac{n}{\alpha})$, for which we need to give a bound of $O(\frac{m}{\alpha})$ time.

Lemma 3.7. *Given a string x of length m, its lexicographically maximum suffix can be found in $O(\frac{m}{\alpha})$ time.*

Proof. Duval's algorithm [28] is an elegant and simple linear time algorithm that can be easily adapted to find the lexicographically maximum suffix. It maintains two positions i and j, one for the currently best suffix and the other for the current candidate. Whenever there is a mismatch after matching k characters ($x[i+k] \neq x[j+k]$), one position is "defeated" and the next candidate is taken. Its implementation in word-RAM is quite straightforward, by comparing α characters at a time, except when the interval $[\min(i,j), \max(i,j)+k]$ contains less than α positions, and so everything stays in a single word: in this case, we can potentially perform $O(\alpha)$ operations for the $O(\alpha)$ characters (contrarily to the rest, where we perform $O(1)$ operations).

We show how to deal with this situation in $O(1)$ time inside a single word w. We employ WSLM on w, and let s be the suffix thus identified in the word w. We set i to be the position of s in the original string x. We set j to be the first occurrence of s in x *after* position i. To compute j, we first do $Z = \text{WSSM}(s, ww')$ where w' is the word immediately following w. By doing `and`(Z, M) with the appropriate bit mask M we can then turn to `0` all bits of Z that correspond to indices i and higher. Finally, j is identified by finding the leftmost `1` bit in `and`(Z, M). We then set k (the number of matches between i and j seen so far) to be $|s|$ and continue by preserving the invariant of Duval's algorithm (i.e., i and j are the best suffix so far and the current candidate). □

Lemma 3.8. *Given a string $x = uv$ of length m, its period $\pi(x)$ can be found in $O(\frac{m}{\alpha})$ time. When $\pi(x) \geq m/2$ and $\pi(x) > \alpha$, we simply report this event without computing $\pi(x)$.*

Proof. We use WSSM where the pattern x' is formed by the first α characters of x and the text by the first $2\alpha - 1$ characters of the second suffix of x. At this point, we take the leftmost occurrence of x' among the first α positions of the second suffix of x. We check for a run of equal $|x'|$-long substrings that are spaced $|x'|$ positions each other in x, starting from the first position in x. If we have that all the positions in x that are multiple of $|x'|$ have occurrences, we know that $\pi(x) \leq |x'| \leq \alpha$; otherwise, it is surely $\pi(x) > \alpha$.

If $\pi(x) \leq \alpha$, the leftmost occurrence of x' in the second suffix of x occurs at position $\pi(x) + 1$ of x. Thus we compute $\pi(x)$ by simple arithmetics.

If $\pi(x) > \alpha$, we first compute the period of v. Observe that the critical factorization $x = uv$ is such that v is lexicographically maximum. We mimic Rytter's linear time code from [54], which finds the period for a string v that is *self-maximal*, that is, lexicographically maximum among its suffixes (as is in our case). This code is reported in Appendix A and can implemented in $O(\frac{m}{\alpha})$ time as required.

Now, if $|u| < |v|$, then the period of v is the local period of the critical factorization $x = uv$, which is the period $\pi(x)$ by definition (see [24]). If $|u| \geq |v|$, we search for v in u as in Section 3.2 since we have all the needed information for the pattern v (e.g. its period). If v occurs in u, we take its rightmost occurrence in u say, at position i, and have $\pi(x) = |u| - i + 1$. If v does not occur in u, we observe that the local period of $x = uv$ is at least $m/2$, and so $\pi(x) \geq m/2$. Since we are dealing with the case $\pi(x) > \alpha$, we simply notify this event. □

Lemma 3.9. *The pre-processing of a pattern of length m takes $O(\frac{m}{\alpha})$ time.*

4. Word-Size Instruction Emulation: Micro-Level Algorithms

Our algorithm in Section 3 uses the two specialized word-size packed string matching instructions, WSSM and WSLM, that are assumed to take $O(1)$ time. In this section we show how to emulate WSSM and WSLM in the standard word-RAM model. Notice that either instructions can be emulated using the Four-Russians trick, this limits the packing factor to $\alpha = \Theta(\log_{|\Sigma|} n)$ and has limited practical value for two reasons: it sacrifices the constant auxiliary space using $O(n^\epsilon)$ words and has no longer has cache-friendly access.

We focus first on the easier and more useful main instruction WSSM that is needed for the text processing and propose efficient bit-parallel emulations in the standard word-RAM, relying on integer multiplication for fast Boolean convolutions and for other operations.

The first emulation requires $\log \omega$ bit padding and thus, larger $\omega \log \omega$ bit words. If only ω bit words are available, one can limit the packing factor to $\alpha / \log \omega$, leading to a $\log \omega$ slowdown.

Lemma 4.1. *The ω-bit* WSSM *instructions can be emulated in $O(1)$ time on a $\omega \log \omega$-bit word-RAM.*

We can reduce the WSSM emulation word size to ω without the above padding and slowdown, but at the cost of a slower pattern pre-processing.

Lemma 4.2. *After a pattern pre-processing that takes $O(\alpha)$ time, the ω-bit* WSSM *instructions can be emulated in $O(1)$ time on a ω-bit word-RAM.*

We then outline the rather standard Four-Russians table lookup emulation for the second instruction WSLM.

Lemma 4.3. *After a pre-processing that takes $O(n^\epsilon)$ time, the ω-bit* WSLM *instructions can be emulated in $O(1)$ time on a ω-bit word-RAM.*

4.1. Bit-parallel emulation of wssm *via Boolean convolutions.*

String matching problems under *general matching relations* were classified in [50, 51, 52] into easy and hard problems, where the easy problems are those equivalent to string matching and are solvable in $O(n+m)$ time, and the hard problems are at least as hard as one or more Boolean convolutions, that are usually solved using the *Fast Fourier Transform (FFT)* and integer convolutions in $O(n \log m)$ time [2, 34]. To efficiently emulate the wssm instruction we introduce *two layers* of increased complexity: first, we recall that the string matching problem can also be solved using Boolean convolutions, and then, we use the powerful, yet standard, integer multiplication operation, that resembles integer convolutions, to emulate Boolean convolutions.[5]

Consider the text $t = t_0 \cdots t_{n-1}$ and the pattern $x = x_0 \cdots x_{m-1}$ where $x_i, t_i \in \Sigma$. Our goal is to compute the occurrence vector of x in t. This is a vector c so that $c_j = \mathtt{1}$ if and only if $t_{j+i} = x_i$ for all $i \in [m]$. Since each character is encoded in $\log_2 |\Sigma|$ bits, we view t and x as *binary* vectors of length $n \log_2 |\Sigma|$ and $m \log_2 |\Sigma|$, respectively, and solve the word-size string-matching problem for binary alphabets. In the occurrence vector c we will then only consider positions $j = 0, \log_2 |\Sigma|, 2 \cdot \log_2 |\Sigma|, \ldots$ that can be obtained using an appropriate bit mask; all the other positions correspond to partial characters and will be discarded. In general, for a binary alphabet, we have that,

$$c_j = \bigwedge_{i=0,\ldots,m-1} (t_{j+i} = x_i) = \overline{\left(\bigvee_{i=0,\ldots,m-1} (t_{j+i} \wedge \overline{x_i}) \right) \vee \left(\bigvee_{i=0,\ldots,m-1} (\overline{t_{j+i}} \wedge x_i) \right)}.$$

Define the *binary convolution operator* $a \star b$ for the binary vectors $a = a_0 \cdots a_{n-1}$ and $b = b_0 \cdots b_{m-1}$ to be:

$$(a \star b)_j = \bigvee_{i=0,\ldots,\min\{m-1,n-j-1\}} (a_{i+j} \wedge b_i).$$

The occurrence vector c can then be computed by taking the n least significant bits from $\overline{(t \star \overline{x}) \vee (\overline{t} \star x)}$. This is illustrated in Figure 2. We now explain how to compute the convolution $t \star \overline{x}$, while computing $\overline{t} \star x$ is done similarly.

Observe that $(t \star \overline{x})_j = \mathtt{1}$ if and only if aligning the pattern x to the text t starting at position j has *at least one* mismatch location where t has a $\mathtt{1}$ and x has a $\mathtt{0}$. We will instead require that $(t \star \overline{x})_j = k$ if and only if there are *exactly* k such mismatches. This way, we can compute $t \star \overline{x}$ using standard integer multiplication $\mathtt{mul}(t, \overline{x})$. This is because with the left shift operator $\mathtt{shl}$ we have:

$$(t \star \overline{x})_j = \bigvee_{i=0,\ldots,\min\{m-1,n-j-1\}} [\mathtt{shl}(t,i) \times \overline{x}_i] = \mathtt{mul}(t, \overline{x})_j.$$

[5] Recall that in the circuit complexity sense Boolean convolution is AC^0, and therefore, is easier than integer multiplication.

The only problem with this method is that the number of mismatches k might be as large as m. To account for this, we pad each digit of t and $\overline{x}$ with $L = \lceil \log m \rceil$ zeros. This is done using the constant time word-RAM padding technique of Fich [32]. We think of each group of $L + 1$ bits as a *field*. Since we are adding up at most m numbers the fields would not overflow. Thus, performing `mul` on the padded strings t and $\overline{x}$ gives fields with value k when the number of mismatches is k.

Adding the two convolutions $t \star \overline{x}$ and $\overline{t} \star x$ together, we get the overall number of mismatches, and we need to identify the fields with value `0` (these correspond to occurrences, i.e., no mismatches). In other words, if we use padded vectors $t', \overline{t'}, x'$, and $\overline{x'}$, we can compute $r = \texttt{add}(\texttt{mul}(t', \overline{x'}), \texttt{mul}(\overline{t'}, x'))$. Our goal is now to set $c_j = \texttt{0}$ if and only if the corresponding field in r is non-zero.

To this end, we first take a bit mask M with `1` in each field at the most significant bit (i.e., $M = (\texttt{10}^L)^\omega$). We then set $X = \texttt{sub}(M, r)$. Notice that a field in r is non-zero if and only if the most significant bit in the corresponding field in X is `0`. Next, we set $Y = \texttt{and}(X, M)$ to keep the most significant bit in each field. Finally, we shift Y right so that in every field the most significant bit moves to the least significant bit. In other words, the desired vector $c = \texttt{shr}(Y, L)$.

Padding the pattern `101` and the text `01101010` (padding bits are in gray) we get that:

$$x = \texttt{010001}, t = \texttt{000101000100010 0}$$

$$\overline{x} = \texttt{000100}, \overline{t} = \texttt{0100000100010001}$$

Doing standard integer multiplication on these vectors we get that:

$$\texttt{mul}(x, \overline{t}) = \texttt{01000101001000100001}$$

$$\texttt{mul}(\overline{x}, t) = \texttt{00000101000100010000}$$

Adding these two numbers we get the mismatch vector:

$$\texttt{add}(\texttt{mul}(x, \overline{t}), \texttt{mul}(\overline{x}, t)) = \texttt{01 00 10 10 00 11 00 11 00 01}$$

Replacing each field by the number it holds gives (two bits, same as base 4):

$$\texttt{add}(\texttt{mul}(x, \overline{t}), \texttt{mul}(\overline{x}, t)) = 1022030301_4$$

Taking the $n = 8$ least significant digits gives the mismatch vector 22030301_4.

Figure 2: An example of searching for the pattern $x = \texttt{101}$ of length $m = 3$ in the text $t = \texttt{01101010}$ of length $n = 8$. The mismatch vector is 22030301. i.e., aligning x to t at position 0 gives two mismatches, at position 1 also gives two mismatches, at position 2 there are no mismatches (this is an occurrence) etc.

The only caveat in the above “string matching via integer multiplication” is its need for padding, thus extending the involved vectors by a factor of $L = \Theta(\log m) = O(\log \omega)$. We now have to use L machine words which incurs a slowdown of $\Omega(L)$. We have thus establisehd Lemma 4.1. In an early version of this work [9], we showed how to reduce the required padding from $\log \omega$ bits to $\log \log \alpha$ bits, using Vishkin's [57] deterministic samples, leading to a smaller $O(\log \log \alpha)$ padding or slowdown; we give next instead the superior constant time emulation first reported in [16].

4.2. Parallel Random-Access-Machine techniques

In our bit-parallel setting, we need an algorithm that easily maps the standard primitives to the word-RAM model. A natural starting point for our quest to find such an algorithm was to look at the existing parallel random-access-machine algorithms. However, such algorithms often make use of the more flexible random memory access that is available, what forced us to come up with an algorithm with a simple and regular memory access pattern, that also greatly simplifies on the published parallel random-access-machine algorithms. The resulting algorithm bares some resemblance to Gąsieniec et al.'s [40] sequential algorithm and to Crochemore et al.'s [23] constant time parallel algorithm. We start by outlining the basic ideas behind the string-matching algorithm, that are based on parallel random-access-machine techniques, and later show how to realize the new algorithm via bit-parallel techniques.

Witness. If the pattern x is not a prefix of zx, namely z is not a period of x, then there must be at least one character mismatch between x and the prefix of zx of length x; such a mismatch is called *a witness* for the non-periodicity of length $|z|$ and it exists for all length $|z|$, such that $1 \leq |z| < \pi(x)$. A witness may be used to eliminate at least one of two close-by occurrences candidates in a process called *duel*, where a text symbol that is aligned with the the witness is used to eliminate at least one of the two occurrence candidates, or both, as that text symbol cannot be equal simultaneously to the two dif and only iferent pattern symbols. Vishkin [56] introduced witnesses in an optimal $O(\log n)$ time parallel string-matching algorithm, improving on an earlier alphabet dependent result by Galil [38]. Breslauer and Galil [14] used witnesses to gain further improvements to optimal $O(\log \log n)$ time and Breslauer and Galil [15] showed that $\Omega(\log \log n)$ time is required by any optimal parallel algorithm that finds the pattern's period length.

Deterministic Sample. A *k-deterministic sample* for the pattern x is a small set DS of locations in x such that if we verify that the pattern occurrence candidate matches all the text symbols aligned at the locations in DS, then no other occurrence candidates that are closer than k locations to each other are plausible; such occurrence candidates are not entirely eliminated by verifying the symbols in DS, but rather must be still sparsified by eliminating candidates that are too close-by to each other. Vishkin [57] introduced deterministic samples in an optimal parallel algorithm that has faster optimal $O(\log^* n)$ time text processing, but slower optimal $O(\log^2 n)$ pattern pre-processing, and proved that there is always a $\pi(x)$-deterministic sample of size $|DS| \leq \log \pi(x)$. Galil [39] improved the text processing to $O(1)$ time and Crochemore et al. [23] used constant-size $\log \log \pi(x)$-deterministic samples to improve the pattern processing time. Ben-Kiki et al. [9] used deterministic samples in the bit-parallel settings, to get a constant time word-size packed string-matching algorithm that uses $O(\omega \log \log \alpha)$ bit words.

Slub. Our approach is based on two stage deterministic samples whose size is not necessarily small, as given by the following definition.

Definition 4.4. *Given a string x of period length $\pi(x)$, we say that the substring z is a* slub *in x if there exist two distinct symbols a and b such that*

1. *both za and zb occur in x, but,*
2. *za occurs only once among the first $\pi(x)$ locations in x.*

Note that za may occur also elsewhere in x among locations $\pi(x)+1, \ldots, |x|$: this is not in contrast with the fact that it occurs once among $1, \ldots, \pi(x)$. Also, if x contains just one distinct symbol, i.e. $x = a^m$ for a symbol a, the string-matching problem is trivially solved. Hence, we assume that $x \neq a^m$ for any symbol a and show next that a slub always exists in x: we actually choose z to be a pattern *prefix*, but other choices are possible. The length of the slub may be anything between 0 and $|x| - 1$.

Lemma 4.5. *If $x \neq a^m$ for any symbol a, there is a prefix z of x that is a slub.*

Proof. Simply let the slub z be the longest prefix of x, such that z occurs at least twice in x, za only occurs at the beginning of x (and possibly also at other locations, which should be after $\pi(x)$) and zb at another location. To see why, recall that a pattern prefix y is called a border of x if it is also a suffix, namely, $x = yy' = y''y$ for two nonempty strings y', y''. It known that any border y of x satisfies $|y| \leq |x| - \pi(x)$, and indeed the longest border matches that equality [47]. Consider now the suffix tree T_x built on x without any terminator: because of this, some internal nodes could be unary, having just one child. Now consider the leaf f in T_x corresponding to x and let z be string stored in the branching node g that is closest ancestor of that leaf f. Note that g always exists as $x \neq a^m$ and the root is branching; moreover, there can be only unary internal nodes along the path from g to f and they *all* correspond to some borders of x longer than z, thus occurring at location 1 and at some other locations after $\pi(x)$. We choose a as the branching character of the edge from g to its child leading to f, and b as the branching character of any other edges from g (and this is always possible since g has at least two children). Note that za occurs only once in the first $\pi(x)$ locations, actually in location 1, and the other occurrences are aligned with x's borders that occur after location $\pi(x)$ since they correspond to the unary nodes along the path from c to f. □

For example, the pattern $a^n b$ has a slub $z = a^{n-1}$ that appears at its beginning followed by the symbol a and starting at the second position followed by the symbol b. The pattern ab^n has a slub that is the empty string $z = \epsilon$ that appears at its beginning followed by the symbol a and starting at all other positions followed by the symbol b.

Lemma 4.6. *Given a slub z and symbols a and b, the deterministic sample consisting of the locations of the symbols a and b in x is a $|z|+1$-deterministic sample of size* 2.

Proof. Observe that any occurrence candidate starting fewer than $|z|+1$ locations *after* an occurrence candidate with a verified deterministic sample $\{a, b\}$ may be eliminated, since a character in z that is aligned with a in one z instance must match b in the other z instance, leading to non-constructive evidence that such an occurrence candidate may be eliminated. See Figure 3-a. □

Goldberg and Zwick [41] also used larger deterministic samples, but our new algorithm is unusual in that it uses only two "simple structure" rather than "small size" deterministic samples, i.e., the first deterministic sample is very small and eliminates occurrence candidates via non-constructive evidence, while the second deterministic sample is a potentially very long prefix of the pattern.

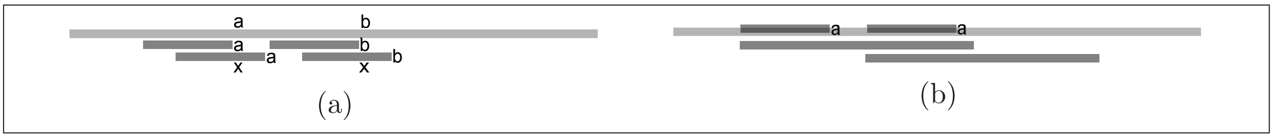

Figure 3: The new string-matching algorithm using slubs: (a) sparsify to one candidate in each $|z| + 1$ block: after verifying the size 2 deterministic sample $\{a, b\}$, any surviving occurrence candidate that has another close-by candidate on its left, can be eliminated. (b) sparsify to one candidate in each period length $\pi(x)$ block: after verifying the deterministic sample za, any surviving occurrence candidate that has another close-by candidate on its right, can be eliminated.

The candidate-based algorithm. Let $x \neq a^m$ be the input pattern with period length $\pi(x)$—and so $x = p^r p'$ where p is the period of x and $|p'| < |p|$ —and z be its slub as in Lemma 4.5. Slubs may be used to obtain a simple parallel string matching algorithm that works by starting out from all the n text positions as candidates:

1. For each occurrence candidate, verify the size-2 deterministic sample for a and b. In each block of length $|z| + 1$, eliminate all remaining occurrence candidates except for the *leftmost* viable candidate by Lemma 4.6. Note that $O(n/(|z| + 1))$ candidates remain.
2. For each remaining occurrence candidate, verify the pattern prefix za. In each block of length $\pi(x)$ eliminate all the remaining occurrence candidates except for the *rightmost* viable candidate since za is unique within the first $\pi(x)$ pattern locations. Note that $O(n/\pi(x))$ candidates survive. See Figure 3-b.
3. For each surviving occurrence candidate, verify the pattern prefix p of length $\pi(x)$. All the occurrences of the period p of x are known at this point.
4. Count periodic runs of p; verify pattern tail p' (a prefix of p) too if it is last in the current run.

Theorem 4.7. *There exist a word-size string-matching algorithm that takes $O(1)$ time in the ω-word-RAM, following pattern pre-processing.*

The proof of Theorem 4.7 is in the implementation of the steps 1–4 of the basic algorithm via bit-parallel techniques as described next. We assume without loss of generality that the pattern x and the text y are binary strings of length at most ω with only `0` and `1` symbols. The pattern pre-processing is described at the end of this section and finds the *period length* $\pi(x)$ of the pattern and the *slub* z with deterministic sample $\{a, b\}$.

Step 1. We first need to verify the small deterministic sample for the slub z. Let $i_a < i_b$ be the indices of the deterministic sample symbols a and b in the pattern x. Without loss of generality suppose that $a = \mathtt{1}$ and $b = \mathtt{0}$. We copy x and complement it into x'. We then perform a left shift `shl` of x' by $i_b - i_a$ locations, so as to align the symbols a and b that need to be verified. We then perform a bit-wise `and` followed by a left shift `shl` by i_a locations to obtain s, so that the resulting `1`s in s mark the remaining candidate occurrences.

At this point, we need to $(|z| + 1)$-sparsify these candidate occurrences in s. We conceptually divide s into blocks of length $|z|+1$ and keep only the leftmost candidate in each block using the bit-wise `lmo` operation. Note that since candidates in consecutive blocks might not be $|z| + 1$ apart, we consider odd and even blocks separately and remove candidates that have a near neighbor. To do this, we employ a binary mask $\mathtt{1}^{|z|+1}\mathtt{0}^{|z|+1}\mathtt{1}^{|z|+1}\mathtt{0}^{|z|+1}\ldots$ in bit-wise `and` with s for odd blocks, and similar for even blocks, and make a bit-wise `or` of the outcomes, storing it into s. As a result, the remaining candidates in s are now represented by `1`s separated by at least $|z|$ `0`s.

Step 2. We can now verify the sparse candidate occurrences marked in s against za, namely, check if each of these candidate occurrence starts with the prefix za, when the candidates are at least $|z| + 1$ locations apart. We again proceed by odd and even blocks, so let us assume without loss of generality that s contains only the candidate occurrences in its odd blocks to avoid interference with those in the even blocks. Consider the word q which contains za followed by a run of `0`s. If we multiply q and s and store the result in c, we obtain that c has a copy of za in each location corresponding to a marked location of s, while the rest are `0`s. If we perform a right shift `shr` of s by $|z| + 1$ locations and make it in `or` with text y, storing the result in y', we have that each potential occurrence of za in y is also an occurrence in y' but terminated by `1`. Since the bits in s are at least $|z| + 1$ apart and we are considering the candidate occurrences in the odd blocks, we get za at the candidate occurrences without interference in this way. Then, we perform a bit-wise `xor` between c and y', storing the result in d. Consider now an odd block and its next (even) block: for the current candidate in location i, there is an occurrence of za if and only if the nearest `1` is at position $i + |z| + 1$. We conceptually divide d into larger blocks of length $2(|z| + 1)$ and keep only the leftmost candidate in each larger block using the bit-wise `lmo` operation. We then perform a left shift `shl` of d by $|z| + 1$ positions and store the result in s', so now each `1` in s' marks an occurrence of za.

At this point, we need to $\pi(x)$-sparsify these candidate occurrences in s'. This is logically done the same way as the $|z| + 1$ sparsification above, only keeping the rightmost surviving candidate in each block of size $\pi(x)$ through the bit-wise `rmo` operation.

Step 3. We can now verify the sparse candidate occurrences marked in s' again the period p of x, namely, check if each of these candidate occurrence starts with the prefix p, when the candidates are at least $\pi(x)$ locations apart. This step is done the same way as the verification again the prefix za in Step 3, using the pattern period p instead of za.

Step 4. Recall that the pattern is $x = p^r p'$. If $r = 1$, we can also check p' in a similar way we did for p, and complete our task by suitably `and`ing the results. Hence we focus here on the interesting case $r \geq 2$. While general counting is dif and only ificult in our setting, our task is simpler since occurrences mean that the periods are lined up in an arithmetic progress. We first select the candidates in which p^2p' occurs: note that this is a minor variation of what discussed so far for p and pp'. Hence, let t be the word in which each `1` marks an occurrence of p^2p'. We filter the runs of these occurrences by storing in t the bit-wise `or` of two words: the former is obtained by putting t in `and` with its left shift `shl` by $\pi(x)$ positions; the latter is obtained by putting t in `and` with its right shift `shr` by $\pi(x)$ positions. At this point, the word t thus obtained has `1` in correspondence of aligned occurrences of p^2p', and we have runs of `1`s at distance $\pi(x)$ from each other inside each run. All we have to do is to remove the last $r-1$ `1`s of each run in t, since they are shorter than the pattern: the remaining ones are the pattern occurrences. Summing up: counting is not possible, but removing these `1`s is doable.

To avoid interferences we consider blocks of $r \times \pi(x)$ bits and work separately on the odd and even blocks, as already discussed before. First mark the last occurrence of p^2p' inside each run of t. This is done by making the `xor` of t with itself shifted right `shr` by $\pi(x)$ locations, and with the result shifted left `shl` by $\pi(x)$ locations and put in `and` with t, storing the outcome in t'. Now the `1`s in t' correspond to the last occurrence of p^2p' inside each run of t. If we multiply the word $(\mathtt{1}\mathtt{0}^{\pi(x)-1})^{r-1}$ (followed by `0`s) by t', and we store in t'' the result shifted left `shl` by $(r-2) \times \pi(x)$ locations, we capture the last $r-1$ entries of each run, exactly those to be removed. At this point, if we complement t'' and make an `and` with t, storing the result in $\hat{t}$, we obtain that the `1`s in $\hat{t}$ finally correspond to the occurrences of x in y.

It is not dif and only ificult to generalize the above steps from a binary alphabet to an alphabet Σ of larger size.

pre-processing the pattern. To solve the string matching problem, the candidate algorithm requires the period length of the pattern $\pi(x)$, which is computed as in Lemma 3.8, and certain deterministic samples on the pattern's *slub*, which is computed in $O(\alpha)$ time as described next. For each of the α pattern suffixes x' of x, we perform $\mathtt{xor}(x, x')$ where x' is padded with $|x| - |x'|$ `0`s, and find the leftmost mismatch in the result, if any, using `lmo`. Either x and x' have a mismatch or x' is a border of x. This takes $O(1)$ time per suffix. It suffices to take the suffix x' whose corresponding mismatch is the farthest from the beginning of x', say at position ℓ: the slub is given by the first ℓ symbols in x and the mismatching symbols are $a = x[\ell]$ and $b = x'[\ell]$. This establishes Lemma 4.2.

Note that the pattern pre-processing is the more dif and only ificult part in parallel string matching algorithms and also in our case using bit-parallelism: as demonstrated by Breslauer and Galil's [15] $\Omega(\log\log n)$ period computation lower bound and the faster text processing by Vishkin [57] and Galil [39]. Our the pre-processing is done in $O(m/\alpha + \alpha)$ time by a purely sequential algorithm, with additional $O(\frac{m}{\log_{|\Sigma|} n})$ for WSLM emulation. Observe that the period for a binary word may be computed in the ω-word-RAM model in constant time using larger $O(\omega \log \omega)$ bit words, or in $O(\log \omega)$ time using $O(\omega)$ bit words.

Remarks on the candidate algorithm in the PRAM

The new algorithm presented in Section 4.2 offers a much simpler constant time concurrent-read concurrent-write (CRCW) parallel random-access-machine algorithm than Galil [39], Crochemore et al. [23] and Goldberg and Zwick [41]. The pattern pre-processing is composed of two parts, the first is the witness computation as in Breslauer and Galil's [14] work, and the second computes the information needed by the new algorithm in this paper in constant time, but only for the pattern prefix of length $\sqrt{m}$. Applying first the algorithm to the pattern prefix of length $\sqrt{m}$ and using periodicity properties, we can sparsify the remaining candidate occurrences to one in every $\sqrt{m}$ block, and then use witnesses to eliminate the rest.

4.3. Four-Russians table lookup technique

We discuss here a simple use of the Four-Russians trick [6] to emulate the WSLM instruction. There is also some circumstantial evidence that WSLM emulation might be harder than the WSSM emulation, since in the parallel random-access-machine model the best lexicographically maximum suffix algorithms take significantly more time than string matching [5, 27, 43].

To emulate the WSLM instruction, we create a lookup table maximum-suffix$[x]$ that gives the length of the lexicographically maximum suffix, for all possible packed strings x represented as integers in base $\log_2 |\Sigma|$ (with their most significant digit as the first character). Using this definition, shorter strings are padded with `0` characters on the left (using the shift right `shr` operation) and therefore have the same maximum suffix as longer strings. This is with the exception of the `0` entry that has the maximum suffix that is the whole string and its length is the length of the whole string which must be specified separately.

The lookup table maximum-suffix is created as follows: Assume we have already computed the length of the maximum suffix of all possible integers with i characters or less. We now want to compute the length of the maximum suffix of all possible integers with $i + 1$ characters where the $i + 1$ character (starting from the right) is some non-zero character $\sigma \in \Sigma$.

Let y be a word with σ as its $i + 1$ character and all other characters `0`. We can obtain y by a shift left operation $y = \texttt{shl}(\sigma, (i + 1) \log_2 |\Sigma|)$. Now, for any integer z with at most i characters we set $x = \texttt{or}(y, z)$. We want to compute the maximal suffix $s(x)$ of x and we already have the maximal suffix $s(z)$ of z. Notice that $s(x)$ is either equal to $s(z)$ or to x itself. We therefore only need to compare the entire string x with $s(z)$ to find which is lexicographically larger. To do so we shift left z until the suffix $s(z)$ starts at the $i + 1$ position. By subtracting this from x we can tell which of x or $s(z)$ is lexicographically larger.

The time to create the maximum-suffix$[x]$ lookup table is clearly linear in its size and the WSLM lexicographically maximum suffix instruction emulation requires one table lookup that takes constant time, establishing Lemma 4.3.

5. Bit-wise Operations: Algorithm for `lmo`

In this section we show how to perform the `lmo` operation that finds the leftmost `1` bits in consecutive blocks inside a word in $O(1)$ time in word-RAM (see Table 3), which is needed when the specialized instruction WSSM and WSLM are not available. Knuth [48] observes that "big-endian and little-endian approaches aren't readily interchangeable in general, because the laws of arithmetic send signals leftward from the bits that are least significant" and therefore assumes that a less traditional bit reversal AC^0 instruction, that can be easily implemented in hardware, is available in his model: such instruction would trivially map between `lmo` and `rmo` operations. Related endian conversion byte reversal instructions, often used in network applications to convert between big and little endian integers, are available in contemporary processors [3, 4, 44, 46].

Given a word $A \in \mathcal{B}$ we are asked to determine the leftmost (the most significant) `1` in each consecutive k-block K_j of A, for $j = 0, \ldots, (\omega/k) - 1$. Each block K_j is formed of contiguous k positions $(j \cdot k) + k - 1, \ldots, (j \cdot k)$. We propose the solution when the size of the consecutive blocks is $k = q^2$, for some integer $q > 0$.

One of the main ideas behind the solution lies in partitioning each block K_j into consecutive sub-blocks $K_j^q, \ldots, K_j^1$, where each K_j^i is of length $q = \sqrt{k}$ and occupies contiguous q positions $(j \cdot k) + i \cdot q - 1, \ldots, (j \cdot k) + (i-1)q$ for $i = 1, 2, \ldots, q$. If a block (resp. sub-block) contains `1`s we say that this is a non-zero block (resp. sub-block.) The `lmo` algorithm operates in three stages:

- During *Stage 1* we identify all non-zero sub-blocks K_j^i.
- Later, in *Stage 2*, we identify in each K_j the leftmost non-zero sub-block $K_j^{i^*}$.
- Finally, in *Stage 3* we identify the leftmost `1` in each sub-block $K_j^{i^*}$ which also refers to the leftmost `1` in K_j.

5.1. Stage 1 (Determine all non-zero sub-blocks)

During this stage we identify all sub-blocks K_j^i in each non-zero K_j. More precisely, for the input word A we compute a word $B \in \mathcal{B}$, in which each non-zero sub-block K_j^i in A, for $i = 1, \ldots, q$ and $j = 0, \ldots, (\omega/k) - 1$, is identified in B by setting the leftmost (most significant) bit $j \cdot k + i \cdot q - 1$ in K_j^i to `1`. The remaining bits in B are set to `0`.

Part A: Extract the leftmost bits in sub-blocks. We will first extract the leftmost bit in each sub-block K_i^j of A. We store extracted bits in a separate word X.

A.1 Create a bit mask M_1= $(\mathtt{1}\mathtt{0}^{q-1})^{\omega/q}$, see Lemma 2.1.

A.2 X= $\mathtt{and}(A, M_1)$, where X contains the leftmost bits extracted from sub-blocks.

Part B: Determine empty (zero) sub-blocks. During this part we create a new word A_1 in which the leftmost bit in each sub-block is set to `1` if the remaining $k - 1$ bits in this sub-block in A are all `0`s. This process is performed as follows:

B.1 Y= $\texttt{and}(\texttt{neg}(A), \texttt{neg}(M_1))$, where Y is obtained from $\texttt{neg}(A)$ by removal of the leftmost bits in each sub-block.

B.2 Create a bit mask M_2= $(\texttt{0}^{q-1}\texttt{1})^{\omega/q}$, see Lemma 2.1.

B.3 A_1= $\texttt{and}(\texttt{add}(Y, M_2), M_1)$, where A_1 is the requested new word.

Part C: Determine non-empty (non-zero) sub-blocks. Finally, we create a word B such that non-zero sub-blocks in A are represented by 1s present at the leftmost bits of these sub-blocks in B and all other bits in B are set to 0.

C.1 A_2= $\texttt{and}(\texttt{neg}(A_1), M_1)$, where A_2 identifies sub-blocks with 1s outside of the leftmost bits.

C.2 B= $\texttt{or}(A_2, X)$.

5.2. Stage 2 (Determine the leftmost non-zero sub-blocks)

The second stage is devoted to computing the leftmost non-zero sub-block $K_j^{i^*}$ in each block K_j (for $j = 0, \ldots, \frac{\omega}{k}-1$) using word B. More precisely, for the input word B (obtained in stage 1) we compute a new word $C \in \mathcal{B}$, in which each of the leftmost non-zero sub-block $K_j^{i^*}$ (for $j = 0, \ldots, \frac{\omega}{k}-1$) is identified in C by the leftmost (most significant) bit $j \cdot k + i^* \cdot q - 1$ in $K_j^{i^*}$ set to 1. The remaining bits in C are set to 0.

In order to make computation of the leftmost non-zero sub-block viable, more particularly to avoid unwanted clashes of 1s, we consider separately three dif and only iferent words based on B. In each of the three words the bits of selected blocks are *culled*, i.e., reset to 0s, as described next.

(1) $B_L = B[K_{\frac{\omega}{k}-1}]\texttt{0}^{2k}B[K_{\frac{\omega}{k}-4}]\texttt{0}^{2k} \ldots B[K_2]\texttt{0}^{2k}$,

(2) $B_M = \texttt{0}^{k}B[K_{\frac{\omega}{k}-2}]\texttt{0}^{2k}B[K_{\frac{\omega}{k}-5}]\texttt{0}^{2k} \ldots B[K_1]\texttt{0}^{k}$, and

(3) $B_R = \texttt{0}^{2k}B[K_{\frac{\omega}{k}-3}]\texttt{0}^{2k}B[K_{\frac{\omega}{k}-6}]\texttt{0}^{2k} \ldots B[K_0]$.

In each of these 3 words, the blocks that are spared from resetting are called *alive blocks*. Without loss of generality (computation on other two words and their alive blocks are analogous) we now show how to compute the most significant non-zero sub-blocks in alive blocks in B_R.

Part D: Fishing out B_R.

D.1 Create a bit mask $M_3 = (\texttt{0}^{2k}\texttt{1}^{k})^{\omega/3k}$, see Lemma 2.1.

D.2 $B_R = \texttt{and}(B, M_3)$, where all bits irrelevant to B_R are culled.

The following lemma holds:

Lemma 5.1. *The only positions at which* `1`*s may occur in B_R refer to the leftmost bits in sub-blocks of alive blocks, i.e., $3j' \cdot k + iq - 1$, for all $j' = 0, \ldots, \frac{\omega}{3k} - 1$ and $i = 1, \ldots, q$.*

Proof. The proof follows directly from the definition of B and M_3. □

Part E: Reversing the leftmost bits of sub-blocks in each block. During this step we multiply and shuffle `1`s of B_R. To be precise, we compute the product $Y = \texttt{mul}(B_R, P)$ of B_R and the shuffling palindrome of the form $P = (\texttt{1}\texttt{0}^q)^{q-1}\texttt{1}$ (i.e., dividing P into segments of q bits each, we obtain all possible shifts of $\texttt{1}\texttt{0}^{q-1}$). The main purpose of this process is to generate a sequence of bits from B_R that appear in the reverse order in Y. These bits can be later searched for the rightmost bit using the procedure `rmo` that already has efficient implementation. We need a couple of more detailed observations.

Lemma 5.2 (Global independence of bits). *In the product of B_R and P there is no interference (overlap) between* `1`*s coming from dif and only iferent alive blocks.*

Proof. Since (alive) blocks with `1`s in B_R are separated by the sequence of `0`s of length $\geq 2k$ and $|P| \leq k$ any `1`s belonging to dif and only iferent alive blocks are simply too far to interfere during the multiplication process. □

Lemma 5.3 (Local independence of bits). *Only one pair of* `1`*s can meet at any position of the convolution vector of the product B_R and P.*

Proof. Observe that bits from the two words meet at the same position in the convolution vector if the sum of the powers associated with the bits are the same. Recall that `1`s in B_R in j'th alive block may occur only at positions $3j' \cdot k + iq - 1$, for $i = 1, \ldots, q$, and all `1`s in P are located at positions $l(q+1)$, for $l = 0, \ldots, q-1$. Assume by contradiction that there exist $1 \leq i_1 < i_2 \leq q$ and $0 \leq l_1 < l_2 \leq q-1$, such that $3j' \cdot k + i_2 q - 1 + l_1(q+1) = 3j' \cdot k + i_1 q - 1 + l_2(q+1)$. The latter equation is equivalent with $(l_2 - l_1)(q+1) = (i_2 - i_1)q$. Since we know that q and $q+1$ are relatively prime and $l_2 - l_1$ and $i_2 - i_1$ are both smaller than q we can conclude that the equivalence cannot hold. □

We focus now our attention on the specific contiguous chunks of bits in Y that contain the leftmost bits of sub-blocks in the same alive block arranged in the reverse order. Consider j'th alive block in B_R.

Lemma 5.4 (Reversed bits). *The bit $3j' \cdot k + iq - 1$, for $i = 1, \ldots, q$, from the j'th alive block in B_R appears in $Y = \texttt{mul}(B_R, P)$ at position $(3j' + 1) \cdot k + q - (i + 1)$.*

Proof. The proof comes directly from the definition of the product of two vectors and the content of B_R and P. In particular, for $i = 1, \ldots q$ the bit $3j' \cdot k + iq - 1$ in B_R meets `1` at position $(q - i)(q + 1)$ in P and it gets replicated at position $3j' \cdot k + iq - 1 + (q - i)(q + 1) =$

$3j' \cdot k + iq - 1 + q^2 + q - iq - i = (3j'+1)k + q - (i+1)$, where we recall that $k = q^2$. $\square$

The sequence of bits $(3j'+1)k + q - (i+1)$ in Y, for $i = 1, \dots q$, and $j' = 0, \dots, \frac{\omega}{3k} - 1$, is called the *$j'$-significant sequence*. The main purpose of the next step is to extract the significant sequences from Y and later to determine the rightmost `1` in each j'-significant sequence. Note that the rightmost `1` in the j'-significant sequence corresponds to the leftmost `1` in the j'th alive block.

Part F: Find the leftmost bits in significant sequences.

F.1 Create bit mask $M_4 = (\mathtt{0}^{3k-q}\mathtt{1}^q)^{\frac{\omega}{3k}}\mathtt{0}^{k-1}$, see Lemma 2.1 where `1`s in this bit mask correspond to the positions located in significant sequences in Y.

F.2 Y_1= $\mathtt{and}(Y, M_4)$, where all positions outside of the significant sequences are culled.

F.3 Y_2= $\mathtt{rmo}(Y_1, 3k)$ where `rmo` operation computes the rightmost bit in each significant sequence of Y_1.

When the rightmost `1` in each significant sequence is determined and stored in Y_2 we apply the shuffle (reverse) mechanism once again. In particular, we use multiplication by palindrome P and shift mechanism to recover the original position of the leftmost `1` (representing the leftmost non-zero sub-block) in each active block of B_R.

Part G: Find the leftmost non-zero sub-block in each active block. Consider the j'th active block and assume that its i^* sub-block is the leftmost non-zero sub-block in this block, where $1 \le i^* \le q$. This sub-block is represented by `1` located at position $3j'k + i^*q - 1$ in B_R. According to Lemma 5.4 after Part E this `1` is present in Y, and in turn in Y_2, at position $(3j'+1)k + q - (i^*+1)$. This time during multiplication of Y_2 by P (step G.1) we focus on the instance of this `1` in the product vector $Y_3 = \mathtt{mul}(Y_2, P)$ generated by the `1` located at position $i^*(q+1)$ in P. This specific instance of `1` from Y_2 is located at position $(3j'+1)k + q - (i^*+1) + i^*(q+1) = (3j'+1)k + q(i^*+1) - 1$. This last expression can be also written as $3j'k + qi^* - 1 + (k+q)$.

We still need to cull unwanted `1`s resulting from multiplication of Y_2 and P. In order to do this, we define a bit mask $M_5 = (\mathtt{0}^{2k}(\mathtt{1}\mathtt{0}^{q-1})^q)^{\frac{\omega}{3k}}\mathtt{0}^{k+q}$ (Steps G.2) in which `1`s are located at positions $3j'k + qi - 1 + (k+q)$, for $j' = 0, \dots \frac{\omega}{3} - 1$ and $i = 1, \dots, q$. And we apply this mask to vector Y_3 (Step G.3). Note first that there must be `1` in M_5 located at position $3j'k + qi^* - 1 + (k+q)$ for any choice of i^*. We need to show, however, that other `1`s in Y_3 obtained on the basis of `1`s located at position $3j'k + qi^* - 1$ in B_R and `1`s in P at positions $(q+1)i$, for $i \ne i^*$, will not survive the culling process. Assume for contradiction that there exists a `1` in Y_3 formed on the basis of a `1` located at the position $(3j'+1)k + q - (i^*+1)$ in Y_2 and a `1` located at position $(q+1)i$ in P, for $i \ne i^*$. The location of this `1` in Y_3 is $(3j'+1)k + q - (i^*+1) + (q+1)i = 3j'k + qi + (i - i^*) - 1 + (k+q)$. But since $0 < |i - i^*| < q$ the location of this `1` cannot overlap with `1`s in M_5.

G.1 Compute Y_3= $\texttt{mul}(Y_2, P)$, where the bits are shuffled again.

G.2 Compute bit mask M_5= $(\texttt{0}^{2k}(\texttt{10}^{q-1})^q)^{\frac{\omega}{3k}}\texttt{0}^{k+q}$, see Lemma 2.1.

G.3 Y_4= $\texttt{and}(Y_3, M_5)$, where we fish out bits of our interest.

G.4 C_R= $\texttt{shr}(Y_4, k+q)$, where we recreate the original location of bits.

We conclude the process (Step G.4) by shifting Y_4 by $k+q$ positions to the right. This is required to relocate the leftmost 1s in the leftmost non-zero sub-blocks to the correct positions. The computations on sequences B_L and B_M can be performed analogously to form C_L and C_M. We conclude by forming C= $\texttt{or}(\texttt{or}(C_L, C_M), C_R)$ which is the combination of C_L, C_M and C_R.

5.3. Stage 3 (Finding the leftmost 1*s in each block)*

The third stage is devoted to the computation of the leftmost 1 in sub-block $K_j^{i^*}$ in each K_j, for $j = 0, \ldots, \frac{\omega}{k} - 1$ on the basis of C. More precisely, for the input word C (obtained on the conclusion of Stage 2) containing information on location of $K_j^{i^*}$ in each block K_j we compute a word $D \in \mathcal{B}$, in which the leftmost 1 in $K_j^{i^*}$ located at position $j \cdot k + i^* \cdot q + l^*$ in $K_j^{i^*}$ is present in D while the remaining bits in D are set to 0. Note that the selected 1s correspond to the leftmost 1s in the blocks, i.e., they form the solution to our problem. As in Stage 2 we also work on three sequences C_L, C_M and C_R separately.

Part H: Extract all bits of the leftmost non-zero sub-blocks $K_j^{i^*}$. This part is performed in four steps. The first three steps are used to construct sequences of 1s corresponding to sub-blocks of our interest. The last step is used to extract the bits from these sub-blocks.

H.1 W_1= $\texttt{shr}(C_R, q-1)$, where the last bit in the sub-block is turned on.

H.2 W_2= $\texttt{sub}(C_R, W_1)$, where all bits (but the most significant one) in the sub-block are turned on.

H.3 W_3= $\texttt{or}(W_1, W_2)$, where all bits in the sub-block are turned on.

H.4 W_4= $\texttt{and}(B_R, W_3)$, where all bits from the sub-block in B_R extracted to the corresponding positions in W_4.

Recall here that in the j'th alive block, where $j' = 0, \ldots, \frac{\omega}{3k} - 1$, the leftmost non-zero sub-block $K_j^{i^*}$ is formed of positions $3j'k + (i^* - 1)q + l$, for $l = 0, \ldots, q-1$ and $j = 3j'$.

We will use the shuffling mechanism again to reverse the order of the bits in $K_j^{i^*}$, apply $\texttt{rmo}$ procedure, and then reverse the sequence of bits once again to conclude the search for the leftmost 1 in each active block.

Part I: Reverse the bits from the sub-block. We start this part by replicating the bits from the sub-block to obtain q copies with the most significant bits located at distance $q+1$. We later extract the sub-block bits located in the reverse order and we find the rightmost bit in the reversed sequence.

I.1 Create a bit mask P_2= $(\mathtt{0}^q\mathtt{1})^q$, see Lemma 2.1.

I.2 Create a bit mask P_3= $(\mathtt{0}^{q-1}\mathtt{1})^q$, see Lemma 2.1.

I.3 W_5= $\mathtt{mul}(C_R, P_2)$, where q copies of the sub-block are formed in W_5.

I.4 Create a bit mask M_6= $\mathtt{mul}(C_R, P_3)$, this is to highlight positions at which the reverse bits from the sub-block appear.

I.5 W_6= $\mathtt{and}(W_5, M_6)$, where the bits from the sub-block appear now in the reverse order, where all other bits are set to `0`s.

We show now that the word W_6 contains the reversed bits of the sub-block. And indeed, recall that the position of the lth bit in the sub-block in C_R is $3j'k + (i-1)q + l$. When we multiply C_R by P_2 this lth bit meets the $(q-l-1)$th bit in P_2, and gets replicated at position $3j'k + (i-1)q + l + (q-(l+1)) \cdot (q+1) = (3j'+1)k + (i-(l+1))q - 1$ in W_5. These decreasing positions are q apart and the smallest position is $3j'k + i^* - 1$, for $l = q-1$. These decreasing positions will coincide with `1`s present in P_3 that are formed on the basis of the most significant bit in the sub-block $3j'k + (i^*-1)q + q - 1$ and positions $l(q+1)$ of `1`s in P_3, for $l = 0, \ldots, q-1$.

Part J: Determine the leftmost `1` and return it to the original position. In the last part of the algorithm we find the rightmost `1` in each block (which corresponds to the leftmost `1` in the active block) using `rmo` operation, and finally we return this bit to its original location.

J.1 W_7= $\mathtt{rmo}(W_6, 3k)$, where the rightmost `1`s (denoted by l^*s) are found.

J.2 W_8= $\mathtt{mul}(W_7, P_1)$, we perform this multiplication to land the l^*th (leftmost) bit of $K_j^{i^*}$ at position $(3j'+1)k + (i^*-1)q + l^* - 1$.

J.5 $W_9 = \mathtt{shr}(W_8, k-1)$, to bring the bits to their original position $3j' \cdot k + (i^*-1)q + l^*$.

J.3 Create a bit mask $M_7 = \mathtt{mul}(W_1, 1^q)$ with `1`s corresponding to the content of the sub-block $K_j^{i^*}$ containing positions $3j' \cdot k + (i^*-1)q + q - 1, \ldots, (3j'+1)k + (i^*-1)q + -1$.

J.4 D_R= $\mathtt{and}(W_9, M_7)$, to cull all bits outside of the sub-block $K_j^{i^*}$.

In conclusion, we note here that the identified `1` at position $(3j'+1)k + (i^*-1)q + l^* - 1$ refers to the leftmost `1` in the block K_j.

Theorem 5.5. *Operation* `lmo` *can be implemented in* $O(1)$ *time in the word-RAM.*

6. Contemporary Commodity Processors

We conducted benchmarks of the packed string matching instructions that are available in the "Efficient Accelerated String and Text Processing: Advanced String Operations" part of the *Streaming SIMD Extension (SSE4.2) and the Advanced Vector Extension (AVX)* on Intel Sandy Bridge processors [44, 46] and consulted Intel's Optimization Reference Manual [45], both indicate remarkable performance.[6] The instruction *Packed Compare Explicit Length Strings Return Mask (PCMPESTRM, Equal Ordered Aggregation)* produces a bit mask that is suitable for short patterns and the similar but slightly faster instruction *Packed Compare Explicit Length Strings Return Index (PCMPESTRI)* produces only the index of the first occurrence, which is suitable for our longer pattern algorithm. These instructions support WSSM with 8-bit or 16-bit characters and with up to 128-bit long pattern and text strings. To the best of our knowledge, there are currently no WSLM equivalent instructions available.

2	4	8	16	32	64	128
SSECP 4.44	**SSECP** 4.57	UFNDMQ4 4.99	BNDMQ4 4.23	BNDMQ4 3.83	LBNDM 3.91	BNDMQ4 3.71
SKIP 4.80	RF 5.07	**SSECP** 5.00	SBNDMQ4 4.31	BNDMQ6 3.86	BNDMQ4 3.94	HASH5 3.83
SO 4.84	BM 5.33	FSBNDM 5.05	UFNDMQ4 4.31	SBNDMQ4 3.95	SBNDMQ4 3.96	HASH8 3.93
FNDM 4.94	BNDMQ2 5.46	SBNDMQ2 5.08	UFNDMQ6 4.47	SBNDMQ6 3.97	BNDMQ6 3.97	HASH3 3.94
FSBNDM 5.03	BF 5.58	BNDMQ2 5.13	SBNDMQ6 4.57	UFNDMQ4 4.00	HASH5 3.98	BNDMQ6 3.97
			23 **SSECP** 5.00	27 **SSECP** 5.29	39 **SSECP** 4.88	42 **SSECP** 4.73
SSECP 4.28	**SSECP** 4.49	BNDMQ2 4.42	SBNDMQ2 4.08	UFNDMQ2 3.75	SBNDMQ4 3.67	BNDMQ4 3.70
FFS 4.88	SVM1 4.84	SBNDMQ2 4.48	UFNDMQ2 4.08	BNDMQ4 3.79	BNDMQ4 3.72	SBNDMQ4 3.71
GRASPM 4.93	SBNDMQ4 4.85	SBNDM 4.59	SBNDMQ4 4.10	SBNDMQ4 3.80	UFNDMQ4 3.80	HASH5 3.75
BR 5.14	BOM2 4.95	SBNDM2 4.59	SBNDM2 4.13	UFNDMQ4 3.80	BNDMQ2 3.89	UFNDMQ4 3.77
BWW 5.14	EBOM 5.25	UFNDMQ2 4.69	BNDMQ2 4.14	BNDMQ2 3.89	SBNDM2 3.96	HASH8 3.80
		13 **SSECP** 5.00	22 **SSECP** 5.08	35 **SSECP** 4.77	39 **SSECP** 4.77	45 **SSECP** 4.76

Table 4: The fastest SMART algorithms, listed by pattern length, on an English text and on a random 16-character text. The numbers to the left of **SSECP** on the bottom line in each table are the ranks of this algorithm among all evaluated SMART algorithms.

Faro and Lecroq kindly made their extensive *String Matching Algorithms Research Tool (SMART)* available to us. Results of the benchmarks that we put together in SMART are summarized in Figure 4. Algorithm *SSECP*, our implementation that uses raw packed string matching instructions for up to 8-character long patterns, performed among the top algorithms; our packed string matching implementation of the Crochemore-Perrin algorithm performed very well on many combinations of input types and longer patterns. In general,

[6] On recent generation high end Intel Sandy Bridge processors, 2-cycle throughput and 7- or 8-cycle latency [45, §C.3.1], still leaving room for further improvement. We did not evaluate the new AMD processors that also realize the same packed string matching instructions [3, 4]. Implicit length, i.e. null terminated, packed string matching instruction variants are also available.

those other algorithms that skip parts of the text have the greater advantage as the pattern becomes *longer* and the variation of the alphabet characters becomes *larger*. These encouraging preliminary experimental results must be interpreted very cautiously, since on one hand we have implemented the benchmarks quickly with minimal effort to optimize the code, while on the other hand the existing SMART algorithms could benefit as well from packed string matching instructions and from additional handcrafted machine specific optimization; in fact, a handful of the existing SMART algorithms already use other Streaming SIMD Extension instructions. See Faro and Lecroq's paper [30, 31] and their SMART implementation for further details on the above listed SMART algorithms.

7. Conclusions

We demonstrated how to employ word-size string matching instructions to design optimal packed string matching algorithms in the word-RAM, which are fast both in theory and in practice. There is an array of interesting questions that arise from our investigation.

1. Is it possible to further improve our WSSM emulation, including pattern pre-processing, with only ω-bit words? With only AC^0 operations, i.e. no integer multiplication? A related open question is whether the pattern pre-processing can be done faster than $O(\log \log n)$ over small integer alphabets in the parallel random-access-machine [13].
2. Derive Boyer-Moore style algorithms, that skip parts of the text [12, 22, 53, 58] and may be therefore faster on average, using packed string matching instructions.
3. Extend our specialized packed string instruction results to the dictionary matching problem with multiple patterns [1, 8, 22, 53] and to other string matching problems.
4. Find critical factorizations in linear time using equality pairwise symbol comparisons, i.e. no alphabet order. Such algorithms could also have applications in our packed string model, possibly eliminating our reliance on the WSLM instruction.
5. Explore more efficient WSLM emulations. The WSLM instruction might be useful in other string algorithms, e.g. Cole and Hariharan's [21] approximating string matching.
6. Further compare the performance of our new algorithm using hardware packed string matching instructions to existing implementations, e.g. Faro and Lecroq's [30, 31] SMART and the SSE4.2 and AVX platform specific *strstr* in *glibc*.
7. Experiment and optimize our bit-parallel WSSM emulation to confirm whether it may be useful in practice in the case of very small alphabets, e.g. binary alphabets or 2-bit DNA alphabets [29, 55].

Acknowledgments

We are grateful to Simone Faro and Thierry Lecroq for making their SMART framework available.

References

[1] A. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *Comm. of the ACM*, 18(6):333–340, 1975.

[2] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[3] AMD. *AMD64® Architecture Programmers Manual Volume 4: 128-Bit and 256-Bit Media Instructions.* AMD Corporation, 2011.

[4] AMD. *Software Optimization Guide for AMD Family 15h Processors.* AMD Corporation, 2011.

[5] A. Apostolico and M. Crochemore. Optimal Canonization of All Substrings of a String. *Inf. Comput.*, 95(1):76–95, 1991.

[6] V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev. On economic construction of the transitive closure of a directed graph. *Soviet Math. Dokl.*, 11:1209–1210, 1970.

[7] R. A. Baeza-Yates. Improved string searching. *Softw. Pract. Exper.*, 19(3):257–271, 1989.

[8] D. Belazzougui. Worst Case Efficient Single and Multiple String Matching in the RAM Model. In *Proceedings of the 21st International Workshop On Combinatorial Algorithms (IWOCA)*, pages 90–102, 2010.

[9] O. Ben-Kiki, P. Bille, D. Breslauer, L. Gąsieniec, R. Grossi, and O. Weimann. Optimal Packed String Matching. In S. Chakraborty and A. Kumar, editors, *FSTTCS*, volume 13 of *LIPIcs*, pages 423–432. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[10] M. Ben-Nissan and S. T. Klein. Accelerating Boyer Moore searches on binary texts. In *Proceedings of the 12th International Conference on Implementation and Application of Automata (CIAA)*, pages 130–143, 2007.

[11] P. Bille. Fast searching in packed strings. *J. Discrete Algorithms*, 9(1):49–56, 2011.

[12] R. Boyer and J. Moore. A fast string searching algorithm. *Comm. of the ACM*, 20:762–772, 1977.

[13] D. Breslauer, A. Czumaj, D. P. Dubhashi, and F. Meyer auf der Heide. Comparison Model Lower Bounds to the Parallel-Random-Access-Machine. *Inf. Process. Lett.*, 62(2):103–110, 1997.

[14] D. Breslauer and Z. Galil. An optimal $O(\log\log n)$ time parallel string matching algorithm. *SIAM J. Comput.*, 19(6):1051–1058, 1990.

[15] D. Breslauer and Z. Galil. A Lower Bound for Parallel String Matching. *SIAM J. Comput.*, 21(5):856–862, 1992.

[16] D. Breslauer, L. Gąsieniec, and R. Grossi. Constant-Time Word-Size Packed String-Matchingg. In *CPM*, pages 83–96, 2012.

[17] D. Breslauer, R. Grossi, and F. Mignosi. Simple Real-Time Constant-Space String Matching. In R. Giancarlo and G. Manzini, editors, *CPM*, volume 6661 of *Lecture Notes in Computer Science*, pages 173–183. Springer, 2011.

[18] A. Brodnik, P. Miltersen, and I. Munro. Trans-Dichotomous Algorithms Without Multiplication - Some Upper and Lower Bounds. In *Proc. 6th Workshop on Algorithms and Data Structures*, number 955, pages 426–439. 1997.

[19] Y. Césari and M. Vincent. Une caractérisation des mots périodiques. *C.R. Acad. Sci. Paris*, 286(A):1175–1177, 1978.

[20] R. Cole, M. Crochemore, Z. Galil, L. Gąsieniec, R. Hariharan, S. Muthukrishnan, K. Park, and W. Rytter. Optimally fast parallel algorithms for preprocessing and pattern matching in one and two dimensions. In *Proc. 34th IEEE Symp. on Foundations of Computer Science*, pages 248–258, 1993.

[21] R. Cole and R. Hariharan. Approximate string matching: A simpler faster algorithm. *SIAM J. Comput.*, 31(6):1761–1782, 2002.

[22] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proc. 6th International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science, pages 118–132. Springer-Verlag, Berlin, Germany, 1979.

[23] M. Crochemore, Z. Galil, L. Gąsieniec, K. Park, and W. Rytter. Constant-Time Randomized Parallel String Matching. *SIAM J. Comput.*, 26(4):950–960, 1997.

[24] M. Crochemore and D. Perrin. Two-way string-matching. *J. ACM*, 38(3):651–675, 1991.

[25] M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, 1994.

[26] A. Czumaj, Z. Galil, L. Gąsieniec, K. Park, and W. Plandowski. Work-time-optimal parallel algorithms for string problems. In F. T. Leighton and A. Borodin, editors, *STOC*, pages 713–722. ACM, 1995.

[27] J. W. Daykin, C. S. Iliopoulos, and W. F. Smyth. Parallel RAM Algorithms for Factorizing Words. *Theor. Comput. Sci.*, 127(1):53–67, 1994.

[28] J. Duval. Factorizing Words over an Ordered Alphabet. *J. Algorithms*, 4:363–381, 1983.

[29] S. Faro and T. Lecroq. Efficient pattern matching on binary strings. In *Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, 2009.

[30] S. Faro and T. Lecroq. The exact string matching problem: a comprehensive experimental evaluation report. Technical Report 0810.2390, arXiv, Cornell University Library, 2011. `http://arxiv.org/abs/1012.2547`.

[31] S. Faro and T. Lecroq. The exact online string matching problem: A review of the most recent results. *ACM Comput. Surv.*, 45(2):13, 2013.

[32] F. E. Fich. Constant Time Operations for Words of Length w. Technical report, University of Toronto, 1999. `http://www.cs.toronto.edu/~faith/algs.ps`.

[33] N. Fine and H. Wilf. Uniqueness theorems for periodic functions. *Proc. Amer. Math. Soc.*, 16:109–114, 1965.

[34] M. Fischer and M. Paterson. String matching and other products. In *Complexity of Computation*, pages 113–125. American Mathematical Society, 1974.

[35] K. Fredriksson. Faster string matching with super-alphabets. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 44–57, 2002.

[36] K. Fredriksson. Shift-or string matching with super-alphabets. *IPL*, 87(4):201–204, 2003.

[37] M. L. Furst, J. B. Saxe, and M. Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1):13–27, 1984.

[38] Z. Galil. Optimal parallel algorithms for string matching. *Inform. and Control*, 67:144–157, 1985.

[39] Z. Galil. A Constant-Time Optimal Parallel String-Matching Algorithm. *J. ACM*, 42(4):908–918, 1995.

[40] L. Gąsieniec, W. Plandowski, and W. Rytter. Constant-space string matching with smaller number of comparisons: sequential sampling. In *Proc. 6th Symp. on Combinatorial Pattern Matching*, number 937 in Lecture Notes in Computer Science, pages 78–89. Springer-Verlag, Berlin, Germany, 1995.

[41] T. Goldberg and U. Zwick. Faster parallel string matching via larger deterministic samples. *J. Algorithms*, 16(2):295–308, 1994.

[42] D. Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology.* Cambridge University Press, 1997.

[43] C. S. Iliopoulos and W. F. Smyth. Optimal Algorithms for Computing the canonical form of a circular string. *Theor. Comput. Sci.*, 92(1):87–105, 1992.

[44] Intel. *Intel® SSE4 Programming Reference.* Intel Corporation, 2007.

[45] Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* Intel Corporation, 2011.

[46] Intel. *Intel® Advanced Vector Extensions Programming Reference.* Intel Corporation, 2011.

[47] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6:322–350, 1977.

[48] D. E. Knuth. *Combinatorial Algorithms*, volume 4A of *The Art of Computer Programming.* Addison-Wesley Professional, Jan. 2011.

[49] M. Lothaire. *Combinatorics on Words.* Addison-Wesley, Reading, MA, U.S.A., 1983.

[50] S. Muthukrishnan. New results and open problems related to non-standard stringology. In *CPM*, pages 298–317, 1995.

[51] S. Muthukrishnan and K. V. Palem. Non-standard stringology: algorithms and complexity. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing (STOC)*, pages 770–779, 1994.

[52] S. Muthukrishnan and H. Ramesh. String Matching Under a General Matching Relation. *Inf. Comput.*, 122(1):140–148, 1995.

[53] G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In M. Farach-Colton, editor, *CPM*, volume 1448 of *Lecture Notes in Computer Science*, pages 14–33. Springer, 1998.

[54] W. Rytter. On maximal suffixes, constant-space linear-time versions of KMP algorithm. *Theor. Comput. Sci*, 1-3(299):763–774, 2003.

[55] J. Tarhio and H. Peltola. String matching in the DNA alphabet. *Software Practice Experience*, 27:851–861, 1997.

[56] U. Vishkin. Optimal parallel pattern matching in strings. *Inform. and Control*, 67:91–113, 1985.

[57] U. Vishkin. Deterministic sampling - A new technique for fast pattern matching. *SIAM J. Comput.*, 20(1):22–40, 1990.

[58] A. C.-C. Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387, 1979.

Appendix A. Rytter's Algorithm

The following linear time code from Rytter [54] finds all the periods for a string v that is *self-maximal*, that is, lexicographically maximum among its suffixes. When the code assigns a new value to `pi`, this is the length of the current prefix of v, since a smaller value (i.e., a shorter period) would contradict the fact that v is lexicographically maximum.

```
selfmax_period(v) {
   pi = 1, len = length(v);
   for (i = 1; i < len; i++)
      if (v[i] != v[i-pi]) pi = i+1;
   return pi;
}
```

Note that the critical factorization $x = uv$ found by the Crochemore-Perrin algorithm is such that v is lexicographically maximum. Also, any of its prefixes satisfies this property: so we can apply the code above to find its period as well (as an intermediate step during the `for` loop) and how long it extends to the rest of v (when the value of `pi` changes).