Towards a Very Fast Multiple String Matching Algorithm for Short Patterns

Simone Faro[1] and M. Oğuzhan Külekci[2]

[1] Dipartimento di Matematica e Informatica, Università di Catania, Italy
[2] TÜBİTAK National Research Institute of Electronics and Cryptology, Turkey
faro@dmi.unict.it, oguzhan.kulekci@tubitak.gov.tr

Abstract. *Multiple exact string matching* is one of the fundamental problems in computer science and finds applications in many other fields, among which computational biology and intrusion detection. It turns out that short patterns appear in many instances of such problems and, in most cases, sensibly affect the performances of the algorithms. Recent solutions in the field of string matching try to exploit the power of the word RAM model to speed-up the performances of classical algorithms. In this model an algorithm operates on words of length w, grouping blocks of characters, and arithmetic and logic operations on the words take one unit of time. This study presents a first preliminary attempt to develop a filter based exact multiple string matching algorithm for searching set of short patterns by taking benefit from Intel's SSE (streaming SIMD extensions) technology. Our experimental results on small, medium, and large alphabet text files show that the proposed algorithm is competitive in the case of short patterns against other efficient solutions, which are known to be among the fastest in practice.

Keywords: multiple string matching, experimental algorithms, text-processing, short patterns, Streaming SIMD Extensions Technology, SSE

1 Introduction

In this article we consider the *multiple string matching problem* which is the problem of searching for all exact occurrences of a set of r patterns in a text t, of length n, where the text and patterns are sequences over a finite alphabet Σ.

Multiple string matching is an important problem in many application areas of computer science. For instance, in computational biology, with the availability of large amounts of DNA data, matching of nucleotide sequences has become an important application and there is an increasing demand for fast computer methods for analysis and data retrieval, e.g., in metagenomics [16,15], we have a set of short patterns which are the extracted DNA fragments of some species, and we would like to check if they exist in another living organism. Although there are various kinds of comparison tools that provide aligning and approximate matching, most of them are based on exact matching in order to speed up the process.

Another important usage of multiple pattern matching algorithms appears in network intrusion detection systems such as Snort [29] as well as in anti-virus software. Snort is a light-weight open-source NIDS which can filter packets based on predefined rules. If the packet matches a certain header rule then its payload is scanned against a set of predefined patterns associated with the header rule. The number of patterns can be in the order of a few thousands[1]. In all these applications, the speed at which pattern matching is performed critically affects the system throughput and although

[1] Snort version 2.9 contains over 2000 strings

Simone Faro, M. Oğuzhan Külekci: Towards a Very Fast Multiple String Matching Algorithm for Short Patterns, pp. 78–91.
Proceedings of PSC 2013, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05330-0

only a small portion of such rules contains short patterns, it turns out that they sensibly affect the performance of multiple string matching algorithm [31]. Moreover another major performance bottleneck of the regarding solutions to these problems is to achieve high-speed multiple pattern matching required to detect malicious patterns of ever growing sets.

This paper presents the results of a first preliminary attempt to develop a fast and practical algorithm for the multiple exact string matching problem which focuses on sets of short patterns. The algorithm we propose, named Multiple Exact Packed String Matching algorithm (MEPSM for short), is designed using specialized word-size packed string matching instructions based on the Intel streaming SIMD extensions (SSE) technology. It can be seen as an extension of the MSSEF algorithm [20,10] that was designed for searching long patterns and has been evaluated amongst the fastest algorithms when the length of the pattern is greater than 32 characters. Thus in the present note we concentrate on solutions which could be used for searching sets of patterns shorter than 32 characters.

This work presents a preliminary result, meaning that our algorithm is still a *work in progress*. Specifically it obtains competitive results only for patterns with a length between 16 and 32, while much work has to be done for obtaining a fast solution for sets of patterns shorter than 16 characters. This will be the goal of our future work.

In Section 2, we introduce some notations and the terminology we adopt throughout the paper. We survey the most relevant existing algorithms for the multiple string matching problem in Section 3. We then present a new algorithm for the multiple string matching problem in Section 4 and report experimental results under various conditions in Section 5. Conclusions and perspectives are given in Section 6.

2 Notions and Terminology

Throughout the paper we will make use of the following notations and terminology. A string p of length $\ell > 0$ is represented as a finite array $p[0 \ldots \ell - 1]$ of characters from a finite alphabet Σ of size σ. Thus $p[i]$ will denote the $(i+1)$-st character of p, and $p[i \ldots j]$ will denote the *factor* (or *substring*) of p contained between the $(i+1)$-st and the $(j+1)$-st characters of p, for $0 \leq i \leq j < \ell$.

Given a set of r patterns $\mathcal{P} = \{p_0, p_1, \ldots, p_{r-1}\}$, we indicate with symbol m_i the length of the pattern p_i, for $0 \leq i < r$, while the length of the shortest pattern in $\mathcal{P}$ is denoted by m', i.e. $m' = \min\{m_i \mid 0 \leq i < r\}$. The length of $\mathcal{P}$, which consists of the sum of the lengths of the $p_i s$ is denoted by m, i.e. $m = \sum_{i=0}^{r-1} m_i$.

We indicate with symbol w the number of bits in a computer word and with symbol $\gamma = \lceil \log \sigma \rceil$ the number of bits used for encoding a single character of the alphabet Σ. The number of characters of the alphabet that fit in a single word is denoted by $\alpha = \lfloor w/\gamma \rfloor$. Without loss of generality we will assume throughout the paper that γ divides w.

In chunks of α characters, any string p of length ℓ is represented by an array of blocks $P[0 \ldots k-1]$ of length $k = \lceil \ell/\alpha \rceil$. Each block $P[i]$ consists of α characters of p and in particular $P[i] = p[i\alpha \ldots i\alpha + \alpha - 1]$, for $0 \leq i < k$. The last block of the string $P[k-1]$ is not complete if $(\ell \mod \alpha) \neq 0$. In that case we suppose the rightmost remaining characters of the block are set to zero. Given a set of patterns $\mathcal{P}$, we define $L = \lceil m'/\alpha \rceil - 1$ as the zero-based address of the last α-character block of the shortest pattern in $\mathcal{P}$, whose individual characters are totally composed of the

characters of the pattern without any padding. Actually, if the length of the shortest pattern in $\mathcal{P}$ is a multiple of α, there is no padding in the last α-characters block, and thus, $L = \lceil m'/\alpha \rceil - 1$. In the other cases, L is the index of the block preceding the last one, as the last one is not a complete block, making $L = \lceil m'/\alpha \rceil - 2$.

Although different values of α and γ are possible, in most cases we assume that $\alpha = 16$ and $\gamma = 8$, which is the most common setting while working with characters in ASCII code and in a word RAM model with 128-bit registers, available in almost all recent processors supporting single instruction multiple data (SIMD) operations.

3 Previous Results

Let $\mathcal{P} = \{p_0, p_1, \ldots, p_{r-1}\}$ be a set of r patterns, where pattern p_i has length m_i, for $0 \leq i < r$, and let t be a text of length n. Moreover let $m = \sum_{i=0}^{r-1} m_i$ and let $m' = \min\{m_i \mid 0 \leq i < m\}$ be the length of the shortest pattern of $\mathcal{P}$.

A first trivial solution to the multiple string matching problem consists of applying an exact string matching algorithm for locating each pattern in $\mathcal{P}$. If we use the well–known Knuth-Morris-Pratt algorithm (KMP) [18], whose search phase is linear in the dimension of the text, this solution has an $O(m + rn)$ worst case time complexity. However, in many practical cases it is possible to avoid reading all the characters of the text achieving sub-linear performances on average.

In a computational model, where the matching algorithm is restricted to read all the characters of the text one by one, the optimal complexity of the multiple pattern matching problem is $\mathcal{O}(m + n)$ while the optimal average complexity of the problem is $O(n \log_\sigma(rm')/m')$ [23]. Such complexities were achieved the first time by the well–known Aho-Corasick algorithm [1] and by the Set-Backward-DAWG-Matching (SBDM) algorithm [26,8], respectively. The SBNDM algorithm is based on the suffix automaton that builds an exact indexing structure for the reverse strings of $\mathcal{P}$ such as a factor automaton or a generalized suffix tree. However experimental investigations highlighted that the bottleneck of the SBDM algorithm is the construction time and space consumption of the exact indexing structure. This can be partially avoided by replacing the exact indexing structure by a factor oracle for a set of strings, which is performed in the Set Backward Oracle Matching (SBOM) algorithm [2].

Hashing is an extensively used approach in string matching [17] and also provides a simple and efficient method to design an efficient algorithm for multiple pattern matching with a sub-linear average complexity. It has been used first by Wu and Manber [34] (WM) whose algorithm constructs an index table for blocks of q characters. Their method is incorporated in the *agrep* command [32].

Recently Faro and Lecroq [13] presented an improvement of WM algorithm based on hashing and q-grams which provides good performances in practical cases. Their method is based on the combination of multiple hash functions with the aim of improving the filtering phase, i.e. to reduce the number of candidate occurrences found by the algorithm. They conduct an experimental evaluation to show the efficiency of the method for matching DNA sequences.

In the last two decades a lot of work has been made in order to exploit the power of the word RAM model of computation to speed-up string matching algorithms for a single pattern. In this model, the computer operates on words of length w, thus blocks of characters are read and processed at once. This means that usual arithmetic and logic operations on the words all take one unit of time. Most of the solutions which

exploit the word RAM model are based on the *bit-parallelism* technique or on the *packed string matching* technique.

The bit-parallelism technique [3] takes advantage of the intrinsic parallelism of the bit operations inside a computer word, allowing to cut down the number of operations that an algorithm performs by a factor up to w. Bit-parallelism is particularly suitable for the efficient simulation of nondeterministic automata [7]. The Shift-Or [3] and BNDM [24] algorithms, which are the representatives of this genre, can be easily extended to the multiple patterns case by deriving the corresponding automata from the maximal trie of the set of patterns [33,25]. The resulting algorithms have a $\mathcal{O}(\sigma\lceil m/w\rceil)$-space complexity and work in $\mathcal{O}(n\lceil m/w\rceil)$ and $\mathcal{O}(n\lceil m/w\rceil m')$ worst-case searching time complexity, respectively. Another efficient solution is the MBNDM algorithm [28], which computes a superimposed pattern from the patterns of the input set when using a condensed alphabet of q characters, and performs filtering using the approach of the standard BNDM algorithm.

However, the bit-parallel encoding requires one bit per automaton state, for a total of (at most) $\lceil m/w\rceil$ computer words. Thus, as long as all the automaton states fit in a computer word, bit-parallel algorithms are extremely fast, otherwise their performances degrade as the number of states of the automaton grows. Although there have been efforts to overcome word-size limitation [19,21,5,6], their performances are still not satisfactory to meet the expectation in practice.

In the *packed string matching* technique multiple characters are packed into one larger word, so that the characters can be compared in bulk rather than individually. In this context, if the characters of a string are drawn from an alphabet of size σ, then $\lfloor w/\log\sigma\rfloor$ different characters fit in a single word, using $\lfloor\log\sigma\rfloor$ bits per character. The packing factor is $\alpha = w/\log\sigma$.

The recent study of Ben-Kiki *et al.* [4] reached the optimal $\mathcal{O}(n/\alpha + occ)$-time complexity for single string matching in $\mathcal{O}(1)$ extra space, where occ is the number of occurrences of the searched pattern. From a practical point of view a very recent algorithm by the authors [10], named Exact Packed String Matching algorithm (EPSM) turns out to be the fastest solution in the case of short patterns. When the length of the searched pattern increases, the SSEF [20] algorithm that performs filtering via the SIMD instructions becomes the best solution in many cases [11,14,12].

In the field of multiple pattern matching in [9] the authors introduced a filter based algorithm, named MSSEF, designed for long patterns, and which benefits from computers intrinsic SIMD instructions. The best and worst case time complexities of the algorithm are $\mathcal{O}(n/m)$ and $\mathcal{O}(nm)$, respectively. The gain obtained in speed via MSSEF becomes much more significant with the increasing set sizes. Hence, considering the fact that the number of malicious patterns in intrusion detection systems or anti-virus software is ever growing as well as the reads produced by next-generation sequencing platforms, the proposed algorithm is supposed to serve a good basis for massive multiple long pattern search applications on these areas.

To the best of our knowledge, packed string matching has not been explored before for multiple pattern matching, and MSSEF is the initial study of this genre.

4 A New Multiple Pattern Matching Algorithm

In this section we present a new multiple string matching algorithm for short patterns, named Multiple Exact Packed String Matching algorithm (MEPSM), and which extends the MSSEF multiple pattern matching algorithm designed for long patterns.

Along the same line of the MSSEF algorithm the MEPSM algorithm is based on a filter mechanism. It first searches the text for candidate occurrences of the patterns using a collection of fingerprint values computed in a preprocessing phase from the set of patterns $\mathcal{P}$. Then the text is scanned by extracting fingerprint values at fixed intervals and in case of a matching fingerprint at a specific position, a naive check follows at that position for all patterns which resemble the detected fingerprint value.

MEPSM is designed to be effective on sets of short patterns, where the the upper limit for the length of the shortest pattern of the set is 32 ($m' \leq 32$). The MEPSM algorithm runs in $\mathcal{O}(nm)$ worst case time complexity and use $\mathcal{O}(rm' + 2^\alpha)$ additional space, where we remember that m' is the length of the shortest pattern in $\mathcal{P}$.

In what follows, we first describe in Section 4.1 the computational model we use for the description of our solutions. Then we describe the preprocessing phase and the searching phase of the MEPSM algorithm in Section 4.2 and in Section 4.3, respectively. We conduct a brief complexity analysis of the algorithm in Section 4.4.

4.1 The Model

In the design of our algorithm we use specialized word-size packed string matching instructions, based on the Intel streaming SIMD extensions (SSE) technology. SIMD instructions exist in many recent microprocessors supporting parallel execution of some operations on multiple data simultaneously via a set of special instructions working on limited number of special registers. Although the usage of SIMD has been explored deeply in multimedia processing, implementation of encryption/decryption algorithms, and on some scientific calculations, only in recent years it has been addressed in string matching [20,9,10].

In our model of computation we suppose that w is the number of bits in a word and σ is the size of the alphabet. When the pattern is short we process the text in chunks of ρ characters, where $\rho \leq \alpha$.

In most practical applications we have $\sigma = 256$ (ASCII code). Moreover SSE specialized instructions allow to work on 128-bit registers, thus reading and processing blocks of sixteen 8-bit characters in a single time unit (thus $\alpha = 16$). Our algorithms are allowed to scan the text in block of 4, 8 and 16 characters.

The specialized word-size packed instruction which is used by our algorithm is named pcrcf (*packed cyclic redundancy check fingerprint*).

A cyclic redundancy check (CRC) is an error-detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data. It was first proposed by W. Wesley Peterson during 1961 [27]. A CRC device calculates a short, fixed-length binary sequence, called *check value*, for each block of data to be sent or stored and appends it to the data. The check value is based on the remainder of a polynomial division of their contents.

Thus the check value of a block of data can be seen as a fingerprint of the block and can be used to evaluate the resemblance of two blocks.

Specifically the instruction pcrcf(B, k), computes an α-bit fingerprint r from a k-bit register B. In practical cases r is a 16-bit register, while the value of k could be 16, 32 or 64, depending on the length of the pattern.

The pcrcf(B, k) specialized instruction can be emulated in constant time by using the following sequence of specialized SIMD instructions

(i) $crc_{32} \leftarrow$ _mm_crc32_u16(ac, B) if B is a 16 bit register
$r \leftarrow$ (unsigned short int) crc_{32}

(ii) $crc_{32} \leftarrow$ _mm_crc32_u32(ac, B) if B is a 32 bit register
$r \leftarrow$ (unsigned short int) crc_{32}

(iii) $crc_{32} \leftarrow$ _mm_crc32_u64(ac, B) if B is a 64 bit register
$r \leftarrow$ (unsigned short int) crc_{32}

Specifically these instructions compute a 32-bit register crc_{32} which is the cyclic redundancy check of the k-bit register B. The parameter ac is a CRC additive constant. The instruction starts with the initial value in ac, accumulates a cyclic redundancy check value for B and stores the result in crc_{32}. Then a second instruction is applied in order to downsample the crc_{32} register and get the 16-bit signature of B. In our implementation we simply take the lower 16 bits of crc_{32} by casting it to an unsigned short int.

In the Intel Core i7 processors, the instructions shown above are implemented with a latency of three cycles and a throughput of one cycle.

We are now ready to describe the new multiple string matching algorithm.

4.2 The Preprocessing Phase

Given a set of patterns $\mathcal{P} = \{p_0, p_1, \ldots, p_{r-1}\}$, where pattern p_i has length m_i, let $m' = \min\{m_i \mid 0 \leq i < r\}$ denote the length of the shortest pattern in $\mathcal{P}$, and $L = \lceil m'/\rho \rceil - 1$. The preprocessing phase of the MEPSM algorithm, which is depicted in Figure 1 (on the left), consists in compiling all the possible fingerprint values of the patterns in the input set $\mathcal{P}$ according to all possible alignments with a block of ρ characters. In particular we set

$$\rho = \min\{i \mid 2^{i+1} > m\},$$

getting $\rho = 16$ when $16 \leq m < 32$, $\rho = 8$ when $8 \leq m < 16$ and $\rho = 4$ when $4 \leq m < 8$.

Thus a fingerprint value is computed for each block $p_i[j \ldots j+\rho-1]$, for $0 \leq i < r$ and $0 \leq j \leq \rho L$. The corresponding fingerprint of a block B of α characters is the α bits register returned by the instruction pcrcf(B, k) (where $k = \rho \times 8$).

To this purpose a table F of size 2^α is computed in order to store, for any possible fingerprint value v, the set of pairs (i, j) such that pcrcf$(p_i[j \ldots j + \rho - 1], k) = v$. More formally we have, for $0 \leq v < 2^\alpha$

$$F[v] = \Big\{(i, j) \mid 0 \leq i < r, 0 \leq j \leq \alpha L \text{ and } \mathsf{wsfp}(p_i[j \ldots j + \rho - 1], k) = v\Big\}.$$

4.3 The Searching Phase

Let t be a text of length n and let $T[0 \ldots N]$ be the text t represented in blocks of ρ characters, where $N = \lceil n/\rho \rceil - 1$. Moreover let $L = \lceil m'/\rho \rceil - 1$.

The basic idea of the searching phase is to compute a fingerprint value for each block of the text $T[zL]$, where $0 \leq z < \lfloor N/L \rfloor$, to explore if it is appropriate to observe any pattern in $\mathcal{P}$ involving an alignment with the block $T[zL]$. If the fingerprint value

```
PREPROCESSING(P, r, m', ρ)
1.  L ← ⌈m'/ρ⌉ − 1
2.  for v ← 0 to 2^α − 1 do F[v] ← ∅
3.  for i ← 0 to r − 1 do
4.      for j ← 0 to ρL do
5.          a ← p_i[j ... j + ρ − 1]
6.          v ← pcrcf(a, ρ × 8)
7.          F[v] ← F[v] ∪ {(i, j)}
8.  return F
```

```
MEPSM(P, r, t, n, ρ)
1.  m' ← min{m_i | 0 ≤ i < r}
2.  F ←Preprocessing(P, r, m', ρ)
3.  N ← ⌈n/ρ⌉ − 1; L ← ⌈m'/ρ⌉ − 1
4.  for s = 0 to N step L do
5.      v ← pcrcf(T[s], ρ × 8)
6.      for each (i, j) ∈ F[v] do
7.          if p_i = t[sρ − j ... sρ − j + m_i − 1] then
8.              output (sρ − j, i)
```

Figure 1. The pseudo-code of the MSSEF multiple string matching algorithm.

indicates that some of the alignments are possible, then those fitting are naively checked.

The pseudo-code given in Figure 1 (on the right) depicts the skeleton of the MEPSM algorithm. The main loop investigates the blocks of the text T in steps of L blocks. If the fingerprint v computed on $T[s]$ is not empty, then the appropriate positions listed in $F[v]$ are verified accordingly.

In particular $F[v]$ contains a linked list of pairs (i, j) marking the pattern p_i and the beginning position of the pattern in the text. While investigating occurrences on $T[s]$, if $F[v]$ contains the couple (i, j), this indicates the pattern p_i may potentially begin at position $(s\rho - j)$ of the text. In that case, a complete verification is to be performed between p and $t[s\rho-j \ldots s\rho-j+m_i-1]$ via a symbol-by-symbol inspection.

4.4 Complexity Analysis

In this Section we give a brief time and space analysis of the MEPSM algorithm.

The preprocessing phase of the MSSEF algorithm requires some additional space to store the rm' possible alignments in the 2^α locations of the table F. Thus the space requirements of the algorithm is $\mathcal{O}(rm' + 2^\alpha)$. Assume $L = \lceil m'/\rho \rceil - 1$. The first loop of the preprocessing phase just initializes the table F, while the second for loop is run $L\alpha$ times. Thus, time complexity of preprocessing is $\mathcal{O}(L\rho)$ that approximates to $O(m)$.

Assume now $N = \lceil n/\rho \rceil - 1$. The searching phase of the algorithm investigates the N blocks of the text T in steps of L blocks. The total number of filtering operations is exactly N/L. At each attempt, the maximum number of verification requests is ρL, since the filter gives information about that number of appropriate alignments of the patterns.

On the other hand, if the computed fingerprint points to an empty location in F, then there is obviously no need for verification. The verification cost for a pattern $p_i \in \mathcal{P}$ is assumed to be $\mathcal{O}(m_i)$, with the brute-force checking of the pattern. Hence, in the worst case the time complexity of the verification is $\mathcal{O}(L\rho m)$, which happens when all patterns in $\mathcal{P}$ must be verified at any possible beginning position.

From these facts, the best case complexity is $\mathcal{O}(N/L)$, and worst case complexity is $\mathcal{O}((N/L)(L\rho m))$, which approximately converge to $\mathcal{O}(n/m')$ and $\mathcal{O}(nm)$ respectively.

5 Experimental results

In this section we present experimental results in order to evaluate the performances of the newly presented algorithm and to compare it against the best algorithms known in literature for multiple string matching problem.

In particular we compared the performances of the MEPSM algorithm against the fastest algorithms known in literature, and specifically:

- MBNDM(q): the Multiple Backward DAWG Matching algorithm [30,28], with values of q ranging from 3 to 8;
- WM(q, h): the Wu-Manber algorithm [34] with, values of q ranging from 3 to 8 and values of h ranging from 1 to 3.

However, in our experimental results only the best versions of the MBNDM(q) and WM(q, h) algorithms are reported, indicating the corresponding values of q and h.

All algorithms have been implemented in the C programming language and have been compiled with the GNU C Compiler, using the optimization options -O3. The experiments were executed locally on an MacBook Pro with 4 Cores, a 2 GHz Intel Core i7 processor, 4 GB RAM 1333 MHz DDR3, 256 KB of L2 Cache and 6 MB of Cache L3. Algorithms have been compared in terms of running times, including any preprocessing time, measured with a hardware cycle counter, available on modern CPUs.

For the evaluation, we use a genome sequence, a protein sequence and a natural language text (English language), all sequences of 4MB. The sequences are provided by the SMART research tool[2] and are available online for download. We have generated sets of 10, 100, 1.000 and 10.000 patterns of fixed length ℓ for the tests. In all cases the patterns were randomly extracted from the text.

For each case we reported the mean over the running times of 200 runs. Tables 1, 2, and 3 lists the timings achieved on genome, protein, and english texts, respectively. Running times are expressed in thousands of seconds. We report the mean of the overall running times and (just below) the mean of the preprocessing time. Best times have been boldfaced.

Moreover it is important to notice that, in our experimental results, the value ℓ was made ranging over the values 16 to 32, which is the range where good results have been obtained by the MEPSM algorithm.

When searching sets of shorter patterns, with a length $m < 16$, our CRC filter technique did not obtain competitive performance underlining that additional work must be done in order to achieve better results on very short patterns.

Table 1 shows experimental results obtained by searching a genome sequence. It turns out that in all cases the MEPSM algorithms obtain the best results against previous solutions. It is up to 3 times faster than the second best result. The speed up becomes more sensible when the size of the set of patterns increases, while it slightly decreases when the length of the patterns increases.

The results reported in Table 1 highlight that the CRC filter technique is particularly efficient for DNA data, improving the performances of the MEPSM algorithm.

In Table 2 results obtained by searching a protein sequence are reported. In this case the MEPSM algorithm obtains always better results only when searching for set of 100 and 1.000 patterns.

[2] The SMART tool is available online at `http://www.dmi.unict.it/~faro/smart/`

(A) m	16	18	20	22	24	26	28	30	32
WM(5,1)	5.64 0.43	5.22 0.42	4.94 0.44	4.70 0.42	4.54 0.43	4.40 0.43	4.31 0.44	4.18 0.43	4.10 0.44
MBNDM(5)	4.42 0.17	4.44 0.17	4.44 0.17	4.45 0.17	4.44 0.17	4.42 0.17	4.45 0.17	4.44 0.17	4.45 0.17
MEPSM	**3.89** 0.01	**3.90** 0.01	**3.89** 0.01	**3.88** 0.01	**3.13** 0.01	**3.12** 0.01	**3.13** 0.01	**3.14** 0.01	**2.78** 0.01

(B) m	16	18	20	22	24	26	28	30	32
WM(5,1)	9.78 0.44	9.38 0.44	8.96 0.44	8.73 0.44	8.60 0.44	8.42 0.44	8.22 0.44	8.10 0.43	8.07 0.44
WM(8,1)	9.98 0.44	8.96 0.44	8.18 0.44	7.62 0.44	7.21 0.44	6.88 0.45	6.55 0.44	6.32 0.44	6.18 0.45
MBNDM(5)	9.04 0.21	9.02 0.21	9.03 0.22	9.03 0.21	9.05 0.21	9.02 0.21	9.05 0.21	9.05 0.21	9.01 0.21
MEPSM	**4.27** 0.04	**4.26** 0.04	**4.24** 0.04	**4.23** 0.04	**3.54** 0.08	**3.54** 0.08	**3.54** 0.08	**3.54** 0.08	**3.24** 0.12

(C) m	16	18	20	22	24	26	28	30	32
WM(8,1)	41.44 0.52	38.26 0.52	37.24 0.54	36.00 0.56	35.08 0.57	34.24 0.59	32.79 0.58	32.42 0.60	32.09 0.60
WM(8,2)	41.55 0.64	32.83 0.69	28.83 0.71	27.56 0.73	25.55 0.73	24.93 0.77	23.02 0.75	22.52 0.78	22.05 0.80
MBNDM(8)	25.22 0.39	25.23 0.39	25.28 0.39	25.09 0.39	25.36 0.39	25.05 0.40	25.14 0.39	25.28 0.40	25.33 0.40
MEPSM	**8.08** 0.40	**8.09** 0.40	**8.05** 0.40	**7.87** 0.40	**7.96** 0.77	**7.92** 0.77	**7.89** 0.77	**7.96** 0.78	**8.53** 1.15

(D) m	16	18	20	22	24	26	28	30	32
WM(5,2)	119.29 1.72	119.49 1.86	119.68 2.00	122.00 2.18	120.50 2.32	120.28 2.43	120.53 2.57	120.82 2.74	120.94 2.85
WM(8,2)	135.98 1.58	126.30 1.77	124.21 2.00	125.15 2.24	123.64 2.43	123.50 2.61	123.78 2.82	123.62 3.04	123.99 3.22
MBNDM(8)	377.14 1.34	386.98 1.37	389.70 1.37	393.60 1.38	393.82 1.39	397.11 1.40	415.40 1.45	421.28 1.46	420.84 1.46
MEPSM	**50.97** 3.87	**51.55** 3.92	**47.62** 3.97	**47.60** 4.00	**51.52** 7.65	**51.90** 7.66	**55.93** 8.21	**54.60** 8.02	**64.85** 11.78

Table 1. Experimental results on a genome sequence of 4 MB. Running times obtained while searching for sets of (A) 10 patterns, (B) 100 patterns, (C) 1.000 patterns and (D) 10.000 patterns.

When the set of patterns is small (10 patterns) the MEPSM algorithm is outperformed by the MBNDM algorithm for short patterns. However it obtains the best results for patterns with a length greater than 22. Again the MBNDM algorithm is the best choice when the set of patterns increases to 10.000 elements. In this last case the performances of the algorithm decreases sensibly when the length of the pattern increases. This behavior is also due to the increase of the preprocessing time consumed by the algorithm.

It turns out from experimental data shown in Table 2 that protein sequences are much more difficult to be filtered by the CRC filter technique proposed in this paper.

Finally in Table 3 experimental results are reported showing the running times obtained by searching on a natural language text. When searching this type of data the MEPSM algorithm turns out to be the best solution in almost all cases. It is second

(A) m	16	18	20	22	24	26	28	30	32
WM(3,1)	5.10 0.44	4.80 0.43	4.58 0.44	4.39 0.43	4.24 0.43	4.12 0.44	4.02 0.43	3.93 0.43	3.87 0.43
WM(6,1)	5.53 0.43	5.10 0.43	4.75 0.44	4.51 0.43	4.33 0.43	4.18 0.43	4.07 0.43	3.93 0.42	3.84 0.42
MBNDM(3)	**3.31** 0.17	**3.30** 0.17	**3.32** 0.17	**3.31** 0.17	3.30 0.17	3.31 0.17	3.31 0.17	3.30 0.17	3.30 0.17
MEPSM	3.89 0.01	3.86 0.01	3.88 0.01	3.88 0.01	**3.14** 0.01	**3.14** 0.01	**3.11** 0.01	**3.12** 0.01	**2.77** 0.01

(B) m	16	18	20	22	24	26	28	30	32
WM(3,1)	5.59 0.43	5.29 0.43	5.07 0.43	4.89 0.43	4.77 0.45	4.72 0.43	4.51 0.43	4.48 0.44	4.36 0.43
WM(4,1)	6.31 0.43	5.93 0.43	5.66 0.43	5.36 0.44	5.06 0.44	5.09 0.45	4.72 0.44	4.64 0.44	4.50 0.44
MBNDM(5)	4.34 0.26	4.34 0.26	4.34 0.26	4.34 0.25	4.34 0.25	4.36 0.26	4.35 0.25	4.35 0.26	4.36 0.26
MEPSM	**4.00** 0.04	**3.99** 0.04	**4.01** 0.04	**4.03** 0.04	**3.33** 0.08	**3.33** 0.08	**3.34** 0.08	**3.34** 0.08	**3.02** 0.12

(C) m	16	18	20	22	24	26	28	30	32
WM(4,1)	8.28 0.53	7.71 0.54	7.49 0.56	7.23 0.56	6.98 0.57	6.86 0.57	6.72 0.59	6.59 0.59	6.48 0.60
WM(8,1)	9.87 0.53	8.78 0.54	8.06 0.56	7.54 0.58	7.11 0.58	6.77 0.59	6.52 0.60	6.31 0.62	6.17 0.64
MBNDM(5)	8.47 0.37	8.52 0.39	8.50 0.38	8.51 0.38	8.57 0.38	8.58 0.39	8.51 0.38	8.49 0.38	8.64 0.39
MBNDM(8)	7.76 0.52	7.81 0.52	7.80 0.53	7.84 0.52	7.84 0.53	7.90 0.54	7.81 0.53	7.85 0.53	7.98 0.54
MEPSM	**5.65** 0.40	**5.67** 0.40	**5.96** 0.40	**6.04** 0.41	**5.63** 0.79	**5.62** 0.79	**5.60** 0.79	**5.61** 0.79	**5.76** 1.18

(D) m	16	18	20	22	24	26	28	30	32
WM(8,1)	24.36 1.54	23.05 1.65	22.48 1.78	22.11 1.91	21.82 2.03	21.76 2.16	21.58 2.25	21.53 2.39	21.63 2.49
MBNDM(8)	**19.75** 1.51	**19.68** 1.51	**19.75** 1.51	**19.76** 1.52	**19.94** 1.55	**19.95** 1.56	**20.06** 1.59	**20.60** 1.60	**20.72** 1.62
MEPSM	22.74 4.05	22.84 4.08	27.43 3.97	27.36 3.95	31.75 7.71	31.29 7.72	32.03 7.74	33.41 7.83	42.73 11.56

Table 2. Experimental results on a protein sequence of 4 MB. Running times obtained while searching for sets of (A) 10 patterns, (B) 100 patterns, (C) 1.000 patterns and (D) 10.000 patterns.

to the WM(q, h) algorithm only in the case of large set of long patterns ($r = 10.000$ patterns and $m \geq 22$). In all other cases the algorithms perform better than previous solutions obtaining a speed up of almost 40% in particular cases.

Table 4 summarizes the speed up ratios achieved via the new algorithms. Here a ratio equal to a value x means that the MPESM algorithm is x times faster than the best solution obtained by a previous algorithm. Thus the larger the ratios mean the better the results, while ratios less than 0 mean that the MPESM algorithm is outperformed by another algorithm.

As can be viewed from that table, the newly proposed solution are in general faster then the competitors. The most significant performance enhancement is observed on

(A) m	16	18	20	22	24	26	28	30	32
WM(5,1)	5.91 0.42	5.47 0.43	5.14 0.43	4.88 0.43	4.71 0.43	4.54 0.43	4.40 0.43	4.29 0.43	4.20 0.44
WM(6,1)	5.85 0.43	5.39 0.43	5.05 0.43	4.77 0.42	4.64 0.42	4.46 0.43	4.30 0.43	4.20 0.42	4.12 0.44
MBNDM(5)	4.37 0.17	4.37 0.17	4.41 0.17	4.39 0.17	4.39 0.17	4.42 0.17	4.43 0.17	4.41 0.17	4.39 0.17
MEPSM	**3.86** 0.01	**3.87** 0.01	**3.87** 0.01	**3.85** 0.01	**3.13** 0.01	**3.12** 0.01	**3.13** 0.01	**3.11** 0.01	**2.77** 0.01

(B) m	16	18	20	22	24	26	28	30	32
WM(5,1)	7.95 0.42	7.44 0.44	6.88 0.42	6.67 0.44	6.34 0.43	6.10 0.44	5.91 0.44	5.80 0.44	5.60 0.44
WM(7,1)	8.37 0.39	7.58 0.41	7.02 0.41	6.66 0.40	6.26 0.40	6.00 0.40	5.80 0.41	5.63 0.40	5.47 0.40
MBNDM(5)	8.22 0.24	8.20 0.25	8.08 0.25	8.17 0.25	8.20 0.25	8.19 0.24	8.21 0.25	8.20 0.25	8.21 0.25
MBNDM(8)	7.48 0.28	7.48 0.28	7.71 0.29	7.56 0.28	7.48 0.28	7.51 0.28	7.48 0.27	7.48 0.27	7.44 0.28
MEPSM	**4.46** 0.04	**4.51** 0.04	**4.45** 0.04	**4.42** 0.04	**3.77** 0.08	**3.71** 0.08	**3.71** 0.08	**3.72** 0.08	**3.40** 0.12

(C) m	16	18	20	22	24	26	28	30	32
WMQ(8,1)	21.01 0.54	18.84 0.55	17.13 0.55	15.96 0.58	15.11 0.59	14.45 0.60	14.04 0.60	13.55 0.62	13.33 0.62
WM(8,2)	27.18 0.67	20.27 0.70	17.41 0.72	15.89 0.76	14.68 0.77	14.07 0.80	13.16 0.82	12.68 0.86	12.24 0.85
MBNDM(5)	16.60 0.38	16.61 0.38	16.57 0.38	16.56 0.38	16.61 0.38	16.54 0.38	16.62 0.38	16.72 0.38	16.65 0.38
MEPSM	**10.37** 0.40	**10.40** 0.40	**9.92** 0.39	**9.93** 0.40	**9.62** 0.79	**9.61** 0.77	**9.61** 0.78	**9.68** 0.78	**9.83** 1.18

(D) m	16	18	20	22	24	26	28	30	32
WM(5,2)	91.42 1.91	86.60 2.03	84.85 2.18	82.93 2.35	83.23 2.52	80.96 2.65	80.34 2.81	79.82 2.94	80.08 3.12
WM(8,2)	90.23 1.70	74.00 1.92	68.27 2.16	**64.27** 2.40	**62.50** 2.65	**59.98** 2.86	**58.69** 3.08	**57.97** 3.32	**57.91** 3.54
MBNDM(8)	116.23 1.45	116.98 1.44	117.73 1.47	118.21 1.47	118.91 1.48	119.29 1.51	119.18 1.50	119.94 1.53	119.78 1.53
MEPSM	**72.54** 3.88	**72.94** 3.87	**66.26** 3.92	67.39 3.97	74.36 7.62	76.10 7.63	75.11 7.55	76.54 7.65	85.21 11.35

Table 3. Experimental results on an english text of 4 MB. Running times obtained while searching for sets of (A) 10 patterns, (B) 100 patterns, (C) 1.000 patterns and (D) 10.000 patterns.

genome sequences, where up to more than 3 fold increase in speed has been observed. Notice that the gain in speed is more significant in the case of a genome sequence and a natural language text.

6 Conclusions

Today, most of the commodity processors are shipped with SIMD instruction sets. Recent studies [20,9,10] benefiting from that technology have been reporting very significant results in pattern matching, where most of the time they outperform their

(A) m	16	20	24	28	32
genome	1.13	1.13	1.41	1.37	1.47
protein	0.85	0.85	1.05	1.06	1.19
nat.lang.	1.13	1.13	1.40	1.37	1.48

(B) m	16	20	24	28	32
genome	2.11	1.92	2.03	1.85	1.90
protein	1.08	1.08	1.30	1.30	1.44
nat.lang.	1.67	1.57	1.66	1.56	1.64

(C) m	16	20	24	28	32
genome	3.12	3.15	3.18	2.91	2.58
protein	1.37	1.37	1.23	1.16	1.07
nat.lang.	1.60	1.67	1.57	1.36	1.24

(D) m	16	20	24	28	32
genome	2.34	2.51	2.33	2.15	1.86
protein	0.86	0.72	0.62	0.62	0.48
nat.lang.	1.24	1.03	0.84	0.78	0.67

Table 4. The speed ups obtained via MEPSM compared with the second best results on sets of 10 (A), 100 (B), 1.000 (C) and 10.000 (D) patterns.

alternatives. This reminds us to consider using SIMD instructions in design and implementation of the algorithms in practice [22].

We have presented a new algorithm targeting patterns shorter than 32 bytes in practice. Experimental results depicted that our proposal becomes a strong alternative to the best known previous solutions when length of the patterns in the set is longer than 16 bytes. We have observed speed ups in orders of magnitudes particularly on genome sequences and English texts as can be seen from Table 4. The CRC filter scales well with the increasing size of the pattern sets.

When the length of the patterns in the set increases, the competitors start scanning quicker as their shift mechanisms improve with longer patterns, and hence, the speed ups compared with our proposal decreases. On patterns shorter than 16 bytes, the CRC filter is not very competitive in its current implementation, and thus, needs further studies to get better results.

In our future work we intend to analyze in depth the impact of the CRC filter in searching sets of short patterns with a length less than 16 characters. We are convinced that good improvements in this direction are possible.

References

1. A. V. Aho and M. J. Corasick: *Efficient string matching: an aid to bibliographic search.* Commun. ACM, 18(6) 1975, pp. 333–340.
2. C. Allauzen, M. Crochemore, and M. Raffinot: *Factor oracle: a new structure for pattern matching*, in Proc. of SOFSEM'99, LNCS 1725, Springer-Verlag, 1999, pp. 291–306.
3. R. Baeza-Yates and G. Gonnet: *A new approach to text searching.* Communications of the ACM, 35(10) 1992, pp. 74–82.
4. O. Ben-Kiki, P. Bille, D. Breslauer, L. Gasieniec, R. Grossi, and O. Weimann: *Optimal packed string matching*, in IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2011), vol. 13 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2011, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 423–432.
5. D. Cantone, S. Faro, and E. Giaquinta: *A compact representation of nondeterministic (suffix) automata for the bit-parallel approach*, in Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, vol. 6129 of Lecture Notes in Computer Science, Springer, 2010, pp. 288–298.
6. D. Cantone, S. Faro, and E. Giaquinta: *A compact representation of nondeterministic (suffix) automata for the bit-parallel approach.* Inf. Comput., 213 2012, pp. 3–12.

7. D. Cantone, S. Faro, and E. Giaquinta: *On the bit-parallel simulation of the nondeterministic aho-corasick and suffix automata for a set of patterns.* J. Discrete Algorithms, 11 2012, pp. 25–36.
8. M. Crochemore and W. Rytter: *Text algorithms*, Oxford University Press, 1994.
9. S. Faro and M. O. Külekci: *Fast multiple string matching using streaming SIMD extensions technology*, in String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, vol. 7608 of Lecture Notes in Computer Science, Springer, 2012, pp. 217–228.
10. S. Faro and M. O. Külekci: *Fast packed string matching for short patterns*, in Proceedings of the 15th Meeting on Algorithm Engineering and Experiments, ALENEX 2013, SIAM, 2013, pp. 113–121.
11. S. Faro and T. Lecroq: *The exact string matching problem: a comprehensive experimental evaluation.* Arxiv preprint arXiv:1012.2547, 2010.
12. S. Faro and T. Lecroq: *2001-2010: Ten years of exact string matching algorithms*, in Proceedings of the Prague Stringology Conference 2011, J. Holub and J. Zdárek, eds., Prague Stringology Club, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2011, pp. 1–2.
13. S. Faro and T. Lecroq: *Fast searching in biological sequences using multiple hash functions*, in 12th IEEE International Conference on Bioinformatics & Bioengineering, BIBE 2012, IEEE Computer Society, 2012, pp. 175–180.
14. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results.* ACM Comput. Surv., 45(2) 2013, p. 13.
15. S. Faro, and E. Pappalardo: *Ant-CSP: An Ant Colony Optimization Algorithm for the Closest String Problem*, in SOFSEM 2010: Theory and Practice of Computer Science, 36th Conference on Current Trends in Theory and Practice of Computer Science, vol. 5901 of Lecture Notes in Computer Science, Springer, 2010, pp. 360–281.
16. S. Gog, K. Karhu, J. Karkkainen, V. Makinen, and N. Valimaki: *Multi-pattern matching with bidirectional indexes*, in Computing and Combinatorics, J. Gudmundsson, J. Mestre, and T. Viglas, eds., vol. 7434 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 384–395.
17. R. M. Karp and M. O. Rabin: *Efficient randomized pattern-matching algorithms.* IBM J. Res. Dev., 31(2) 1987, pp. 249–260.
18. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt: *Fast pattern matching in strings.* SIAM J. Comput., 6(1) 1977, pp. 323–350.
19. M. O. Külekci: *TARA: An algorithm for fast searching of multiple patterns on text files*, in IEEE 22nd International Symposium on Computer and Information Sciences (ISCIS), 2007, pp. 1–6.
20. M. O. Külekci: *Filter based fast matching of long patterns by using SIMD instructions*, in Proceedings of the Prague Stringology Conference, 2009, pp. 118–128.
21. M. O. Külekci: *BLIM: A new bit-parallel pattern matching algorithm overcoming computer word size limitation.* Mathematics in Computer Science, 3(4) 2010, pp. 407–420.
22. S. Ladra, O. Pedreira, J. Duato, and N. Brisaboa: *Exploiting SIMD instructions in current processors to improve classical string algorithms*, in Advances in Databases and Information Systems, T. Morzy, T. Harder, and R. Wrembel, eds., vol. 7503 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 254–267.
23. G. Navarro and K. Fredriksson: *Average complexity of exact and approximate multiple string matching.* Theor. Comput. Sci., 321(2-3) 2004, pp. 283–290.
24. G. Navarro and M. Raffinot: *A bit-parallel approach to suffix automata: Fast extended string matching*, in Combinatorial Pattern Matching, Springer, 1998, pp. 14–33.
25. G. Navarro and M. Raffinot: *Fast and flexible string matching by combining bit-parallelism and suffix automata.* ACM J. Experimental Algorithmics, 5 2000, p. 4.
26. G. Navarro and M. Raffinot: *Flexible pattern matching in strings - practical on-line search algorithms for texts and biological sequences*, Cambridge Univ. Press, 2002.
27. W.W. Peterson and D.T. Brown. *Cyclic Codes for Error Detection.* Proceedings of the IRE, 49 (1): 228–235, 1961.
28. E. Rivals, L. Salmela, P. Kiiskinen, P. Kalsi, and J. Tarhio: *Mpscan: Fast localisation of multiple reads in genomes*, in Proc. of WABI, 2009, pp. 246–260.

29. M. Roesch: *Snort - lightweight intrusion detection for networks*, in Proceedings of the 13th USENIX conference on System administration, LISA '99, Berkeley, CA, USA, 1999, USENIX Association, pp. 229–238.
30. L. Salmela and J. Tarhio: *Multi-pattern string matching with q-grams.* ACM Journal of Experimental Algorithmics, 11 2006, pp. 1–19.
31. B. Zhang, X. Chen, X. Pan and Z. Wu. *High concurrence Wu-Manber multiple patterns matching algorithm.* Proceedings of the International Symposium on Information Proces, p. 404, 2009.
32. S. Wu and U. Manber: *Agrep – a fast approximate pattern-matching tool*, in Proceedings of USENIX Winter 1992 Technical Conference, USENIX Association, 1992, pp. 153–162.
33. S. Wu and U. Manber: *Fast text searching: allowing errors.* Commun. ACM, 35(10) 1992, pp. 83–91.
34. S. Wu and U. Manber: *A fast algorithm for multi-pattern searching*, Report TR-94-17, Dep. of Computer Science, University of Arizona, Tucson, AZ, 1994.