Fast Multiple String Matching Using Streaming SIMD Extensions Technology

Simone Faro[†] and M. Oğuzhan Külekci[‡]

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

Abstract. Searching for all occurrences of a given set of patterns in a text is a fundamental problem in computer science with applications in many fields, like computational biology and intrusion detection systems. In the last two decades a general trend has appeared trying to exploit the power of the word RAM model to speed-up the performances of classical string matching algorithms. This study introduces a filter based exact *multiple* string matching algorithm, which benefits from Intel's SSE (streaming SIMD extensions) technology for searching long strings. Our experimental results on various conditions show that the proposed algorithm outperforms other solutions, which are known to be among the fastest in practice.

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 example, 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. Similarly, in metagenomics [22], we have a set of patterns which are the extracted DNA fragments of some species, and would like to check if they exist in another living organism. Another important usage of multiple pattern matching algorithms appears in network intrusion detection systems as well as in anti-virus software, where such systems should check an increasing number of malicious patterns on disks or high–speed network traffic. The common properties of systems demanding for multi–pattern matching is ever increasing size of both the sets and pattern lengths. Hence, searching of multiple long strings over a sequence is becoming a more significant problem.

In this paper we present a new practical and efficient algorithm for the multiple exact string matching problem that turns out to be faster than the best algorithms known in literature in most practical cases. The algorithm, named

Multiple Pattern Streaming SIMD Extensions Filter (MPSSEF), is designed using specialized word-size packed string matching instructions based on the Intel streaming SIMD extensions (SSE) technology. To the best of our knowledge, MPSSEF is the first algorithm that exploits the power of the word RAM model for the multiple string matching problem. It can be seen as an extension of the SSEF algorithm [12] that was designed for single pattern matching and has been evaluated amongst the fastest algorithms in the case of long patterns [10, 8].

The paper is organized as follows. In Section 2, we introduce some notions and the terminology we adopt along 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 $p[0 \dots \ell - 1]$ of characters from a finite alphabet Σ of size σ. Thus $p[i]$ will denote the $(i+1)$-st character of p, and $p[i \dots j]$ 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 $\mathcal{P} = \{p_0, p_1, \dots, p_{r-1}\}$, 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 along the paper that γ divides w.

In chunks of α characters, any string p of length ℓ is represented by an array of blocks $P[0 \dots k-1]$ of length $k = \lceil \ell/\alpha \rceil$. Each block $P[i]$ consists of α characters of p and in particular $P[i] = p[i\alpha \dots i\alpha + \alpha - 1]$, 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.

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 commodity processors supporting single instruction multiple data (SIMD) operations.

3 Previous Results

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 [11], which is linear in the dimension of the text, this solution has an $O(m+rn)$ worst case time complexity.

The optimal average complexity of the problem is $O(n \log_\sigma(rm')/m')$ [17]. This bound has been achieved by the Set-Backward-DAWG-Matching (SBDM) algorithm [16, 6] 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. Hashing also provides a simple and efficient method, where it has been used first by Wu and Manber [20] to design an efficient algorithm for multiple pattern matching with a sub-linear average complexity which uses an index table for blocks of q characters.

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. 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 [2] 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. The Shift-Or [2] and BNDM [14] 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 [21, 15]. 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 [18], which computes a superimposed pattern from the patterns of the input set when using a condensed alphabet of q characters, and performs filtering with the standard BNDM.

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 characters. The packing factor is $\alpha = w/\log\sigma$.

The recent study of Ben-Kiki *et al.* [3] 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 in the text. The authors showed in their experimental results that their algorithm turns out to be among the fastest solutions in the case of short patterns. When the length of the searched pattern increases, the SSEF [12] algorithm that performs filtering via the SIMD instructions becomes the best solution in many cases [8, 10]. However, to the best of our knowledge, packed string matching has not been explored before for multiple pattern matching, and MPSSEF 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, named Multiple Patterns Streaming SIMD Extension Filter (MPSSEF), which can be viewed a generalization of the SSEF algorithm designed for single string matching. The 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. MPSSEF is designed to be effective on sets of long patterns, where the lower limit for the shortest pattern of the set is 32 ($m' \geq 32$). Although it is possible to adapt the algorithm for lesser lengths, the performance gets worse under 32. The MPSSEF 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 $\mathcal{P}$.

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.

In particular our algorithm makes use of the wsfp (*word-size fingerprint instruction*) specialized word-size packed instruction. The instruction $\mathsf{wsfp}(B, k)$, computes an α-bit fingerprint from a w-bit register B handled as a block of α characters. Assuming $B[0 \ldots \alpha - 1]$ is a w-bit integer parameter, $\mathsf{wsfp}(B, k)$ returns an α-bit value $r[0 \ldots \alpha - 1]$, where $r[i] = 1$ iff the bit at position $\gamma - 1 - k$ in $B[i]$ is set, and $r[i] = 0$ otherwise. The $\mathsf{wsfp}(B, k)$ specialized instruction can be emulated in constant time by using the following sequence of specialized SIMD instructions

$D \leftarrow \_\mathsf{mm\_slli\_epi64}(B, k)$
$r \leftarrow \_\mathsf{mm\_movemask\_epi8}(D)$

Specifically the $\_\mathsf{mm\_slli\_epi64}(B, k)$ instruction individually shifts left the two 64-bit blocks of the 128-bit word B of k positions, and set to zero the rightmost k bits of the first and second halves. Assuming $B = a_1 a_2$, where a_1 and a_2 are the 64-bits long first and second parts of B, this corresponds to the standard shift instruction $(a_1 \ll k)$ and $(a_2 \ll k)$.

The $\_\mathsf{mm\_movemask\_epi8}(D)$ instruction gets a 128 bit parameter D, handled as sixteen 8-bit integers, and creates a 16-bit mask from the most significant bits of the 16 integers in D, and zero extends the upper bits.

```
PREPROCESSING(P, r, m', k)
1. L ← ⌈m'/α⌉ − 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 αL do
5.         a ← p_i[j ... j + α − 1]
6.         v ← wsfp(a, k)
7.         F[v] ← F[v] ∪ {(i, j)}
8. return F
```

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

Fig. 1. The pseudo-code of the MPSSEF multiple string matching algorithm.

4.2 The Preprocessing Phase

The preprocessing phase of the MPSSEF algorithm, which is depicted in Fig. 1 (on the left), consist 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 α characters. Thus a fingerprint value is computed for each block $p_i[j \dots j+\alpha-1]$, for $0 \leq i < r$ and $0 \leq j \leq \alpha L$. The corresponding fingerprint of a block B of α characters is the α bits register returned by the instruction $\mathsf{wsfp}(B, k)$ and formed by concatenating the leftmost bits of each character after shifting by k bits. 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 $\mathsf{wsfp}(p_i[j \dots j+\alpha-1], k) = v$. More formally we have, for $0 \leq v < 2^\alpha$

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

The reason for shifting by k positions is to generate a distinguishing fingerprint value. Such a value must be selected depending on the alphabet size and characters distribution of the text. For example, when the search is to be performed on an English text, the leftmost bits of bytes are generally 0 as in the standard ASCII table the printable characters of the language reside in the first 128 values, where the leftmost bits are always 0. If we do not include a shift operation, then the fingerprint values would be $v = 0^\alpha$ in all cases, and while scanning the text the verification step would be called at each position.

As another example, let's consider pattern matching on an ASCII coded plain DNA sequence, where the alphabet is a, c, g and t, having ASCII codes 01100001, 01110100, 01100011, and 01100111, respectively. The first three bits and the fifth bit are all the same. Since the number of 1s and 0s are equal on the sixth and seventh positions from the remaining bits, one of them, say 6th, may be used as the distinguishing bit. Thus $k = 5$ would be a good choice.

The preprocessing phase of the MPSSEF algorithm requires some additional space to store the rm' possible alignments in the 2^α locations of the table F. Thus, the space requirement of the algorithm is $\mathcal{O}(rm' + 2^\alpha)$. The first loop of the preprocessing phase just initializes the table F, while the second for loop is run $L\alpha$ times, which makes the time complexity of preprocessing $\mathcal{O}(L\alpha)$ that approximates to $O(m)$.

4.3 The Searching Phase

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

The pseudo-code given in Fig. 1 (on the right) depicts the skeleton of the MPSSEF 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 $(s\alpha - j)$ of the text. In that case, a complete verification is to be performed between p and $t[s\alpha - j \ldots s\alpha - j + m_i - 1]$ via a symbol-by-symbol inspection.

The total number of filtering operations is exactly N/L. At each attempt, maximum number of verification requests is αL, 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 $\mathcal{O}(L\alpha m)$, 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 $\mathcal{O}((N/L)(L\alpha m))$, which approximately converge to $\mathcal{O}(n/m')$ and $\mathcal{O}(nm)$ respectively.

4.4 Tuning the MPSSEF algorithm

As stated above, the preprocessing time of the MPSSEF algorithm is $\mathcal{O}(2^{\alpha}+m'r)$ that strongly depends on the size of the set of patterns $\mathcal{P}$ and on the length m' of the shortest pattern in $\mathcal{P}$. This leads to an explosion of the preprocessing time of the algorithm when searching for large sets of long patterns.

Similarly, when the number of pairs stored in the table F increases, the number of verifications called during the searching phase increases proportionally. Thus when searching for a large set of long patterns most of the time spent during the searching time is in the verification step, since the number of pairs stored in F is proportional to r and m'.

An efficient solution to avoid the problems described above, consists in preprocessing the set of patterns computing the fingerprints for prefixes of fixed length $q \leq m'$ instead of for prefixes of length m'. This allows to reduce the preprocessing time to $\mathcal{O}(qr)$ which depends only on the size of the set $\mathcal{P}$. Thus, for a fixed length q, the preprocessing phase consists in computing a table F of

Set Size	Algorithm	32	64	128	256	512
100	MPSSEF	0.08 : 3.14	0.23 : 2.53	0.54 : 2.42	1.16 : 2.35	2.37 : 2.42
	MPSSEF^{64}	0.08 : 3.14	0.23 : 2.53	0.24 : 2.53	0.24 : 2.52	0.25 : 2.53
1.000	MPSSEF	0.82 : 4.78	2.33 : 4.85	5.24 : 5.37	11.24 : 5.82	22.34 : 6.78
	MPSSEF^{32}	0.82 : 4.78	0.81 : 4.44	0.82 : 4.40	0.91 : 4.52	1.00 : 4.55
10.000	MPSSEF	7.65 : 21.46	22.07 : 42.08	52.09 : 51.36	110.25 : 57.53	228.42 : 63.72
	MPSSEF^{32}	7.65 : 21.46	7.81 : 21.10	8.50 : 26.58	9.45 : 27.31	11.12 : 35.17

Table 1. Preprocessing and searching times of the MPSSEF and MPSSEF^q algorithms for searching sets of 100, 1.000 and 10.000 patterns on a genome sequence.

size 2^α in order to store, for any possible fingerprint value v, the set of pairs (i, j) such that $p_i[j \dots j + \alpha - 1] = v$, with $0 \leq j \leq q$. More formally

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

for $0 \leq v < 2^\alpha$, and where we have to choose the parameter q as a multiple of α.

Then the main loop of the searching phase of the algorithm investigates the blocks of the text T in steps of $L = q/\alpha$ blocks. In most cases this reduces the step between two investigated blocks of the text, since in general $q/\alpha \leq \lceil m'/\alpha \rceil - 1$. However, the drop in performances caused by the reduction of the step is offset by the gain in performances due to the reduction of the preprocessing time and the number of verification calls. We name the resulting tuned version of the algorithm as MPSSEF^q algorithm.

Table 1 shows data extracted from Table 2 and puts stress on the preprocessing and searching times of the original MPSSEF algorithm compared with those of the MPSSEF^q algorithm. The running times have been computed on a genome sequence of 4Mb. The details of the experimental results are given in Section 5. In particular the table shows the preprocessing and the searching times of the two algorithms when used for searching set of 100, 1000, and 10000 patterns of equal length ℓ ranging from 32 to 512. The MPSSEF^q algorithm was tuned with $q = 64$ for searching sets of 100 patterns and with $q = 32$ in the other case.

We can notice that the preprocessing time of the MPSSEF algorithm linearly increases with the length of the patterns. A less evident trend can be noticed also in the case of searching times, especially for large sets of patterns. On the other hand the preprocessing and the searching times of the MPSSEF^q algorithm show a linear trend in almost all cases.

5 Experimental results

We compared the performances of the newly presented MPSSEF) and its q-characters filtered version MPSSEF^q against the following best algorithms known in literature for multiple string matching problem: (MBNDM) The Multiple

(A) m	32	64	128	256	512	1024
MBNDMq	$4.27^{(5)}$[0.16]	$4.29^{(5)}$[0.16]	$4.28^{(5)}$[0.16]	$4.34^{(5)}$[0.16]	$4.36^{(5)}$[0.17]	$4.38^{(5)}$[0.17]
WMq	$4.43^{(6)}$[0.40]	$3.58^{(8)}$[0.41]	$3.34^{(8)}$[0.43]	$3.15^{(8)}$[0.43]	$2.98^{(8)}$[0.43]	$2.86^{(8)}$[0.47]
MPSSEF	**2.98**[0.01]	**2.38**[0.02]	**2.27**[0.06]	**2.15**[0.12]	**2.13**[0.24]	2.31[0.48]
MPSSEF^{128}	**2.98**[0.01]	**2.38**[0.02]	**2.27**[0.06]	2.38[0.06]	2.32[0.06]	**2.30**[0.06]
(B)						
MBNDMq	$8.69^{(5)}$[0.20]	$8.74^{(5)}$[0.20]	$8.70^{(5)}$[0.21]	$8.75^{(5)}$[0.22]	$8.78^{(5)}$[0.23]	$8.81^{(5)}$[0.23]
WMq	$6.18^{(8)}$[0.43]	$4.89^{(8)}$[0.44]	$4.38^{(8)}$[0.48]	$4.18^{(8)}$[0.55]	$4.17^{(8)}$[0.67]	$4.45^{(8)}$[0.93]
MPSSEF	**3.23**[0.08]	**2.77**[0.23]	2.96[0.54]	3.51[1.16]	4.78[2.37]	7.13[4.73]
MPSSEF^{64}	**3.23**[0.08]	**2.77**[0.23]	**2.76**[0.24]	**2.76**[0.24]	**2.79**[0.25]	**2.78**[0.27]
(C)						
MBNDMq	$24.78^{(8)}$[0.38]	$24.38^{(8)}$[0.39]	$24.97^{(8)}$[0.43]	$25.32^{(8)}$[0.48]	$25.47^{(8)}$[0.56]	$26.41^{(8)}$[0.82]
WMq	$33.03^{(8)}$[0.61]	$30.80^{(8)}$[0.77]	$30.81^{(8)}$[1.10]	$31.74^{(8)}$[1.76]	$34.45^{(8)}$[3.14]	$36.72^{(8)}$[5.76]
MPSSEF	**5.60**[0.82]	7.19[2.33]	10.61[5.24]	17.06[11.24]	29.12[22.34]	54.64[46.61]
MPSSEF^{32}	**5.60**[0.82]	**5.25**[0.81]	**5.22**[0.82]	**5.44**[0.91]	**5.55**[1.00]	**6.06**[1.28]
(D)						
MBNDMq	$389.5^{(8)}$[1.38]	$393.0^{(8)}$[1.56]	$408.5^{(8)}$[2.03]	$426.9^{(8)}$[2.90]	$443.1^{(8)}$[4.68]	$522.7^{(8)}$[9.02]
WMq	$338.3^{(8)}$[2.19]	$346.1^{(8)}$[3.86]	$357.2^{(8)}$[7.31]	$361.9^{(8)}$[13.65]	$388.0^{(8)}$[26.53]	$460.4^{(8)}$[52.88]
MPSSEF	**29.11**[7.65]	64.15[22.07]	103.45[52.09]	167.79[110.25]	292.14[228.42]	581.09[505.90]
MPSSEF^{32}	**29.11**[7.65]	**28.90**[7.81]	**35.08**[8.50]	**36.76**[9.45]	**46.29**[11.12]	**51.09**[13.04]

Table 2. Running times for 10 (A), 100 (B), 1.000 (C) and 10.000 (D) patterns on a genome sequence.

(A) m	32	64	128	256	512	1024
MBNDMq	$3.24^{(3)}$[0.16]	$3.40^{(3)}$[0.17]	$3.22^{(3)}$[0.16]	$3.20^{(3)}$[0.16]	$3.26^{(3)}$[0.17]	$3.20^{(3)}$[0.16]
WMq	$3.81^{(6)}$[0.41]	$3.26^{(6)}$[0.40]	$3.14^{(6)}$[0.43]	$2.96^{(6)}$[0.42]	$2.88^{(6)}$[0.42]	$2.77^{(6)}$[0.44]
MPSSEF	**2.99**[0.01]	**2.39**[0.03]	**2.23**[0.06]	**2.14**[0.12]	**2.07**[0.24]	2.32[0.48]
MPSSEF^{128}	**2.99**[0.01]	**2.39**[0.03]	**2.23**[0.06]	2.40[0.06]	2.36[0.06]	**2.31**[0.06]
(B)						
MBNDMq	$4.26^{(5)}$[0.25]	$4.27^{(5)}$[0.25]	$4.28^{(5)}$[0.25]	$4.31^{(5)}$[0.26]	$4.33^{(5)}$[0.26]	$4.41^{(5)}$[0.28]
WMq	$4.42^{(4)}$[0.43]	$3.71^{(4)}$[0.44]	$3.41^{(4)}$[0.47]	$3.30^{(4)}$[0.53]	$3.36^{(4)}$[0.64]	$3.45^{(4)}$[0.86]
MPSSEF	**3.23**[0.08]	**2.74**[0.23]	2.92[0.54]	3.48[1.15]	4.65[2.31]	7.09[4.71]
MPSSEF^{64}	**3.23**[0.08]	**2.74**[0.23]	**2.80**[0.24]	**2.84**[0.25]	**2.83**[0.26]	**2.87**[0.28]
(C)						
MBNDMq	$7.67^{(8)}$[0.52]	$7.71^{(8)}$[0.53]	$7.86^{(8)}$[0.56]	$8.10^{(8)}$[0.62]	$8.46^{(8)}$[0.69]	$9.38^{(8)}$[0.96]
WMq	$6.09^{(8)}$[0.62]	**5.16**$^{(8)}$[0.81]	**5.13**$^{(8)}$[1.18]	$5.76^{(8)}$[1.90]	$7.26^{(4)}$[2.67]	$9.94^{(4)}$[4.90]
MPSSEF	5.57[0.81]	7.22[2.35]	10.59[5.23]	16.46[10.94]	29.29[22.50]	54.48[46.64]
MPSSEF^{32}	**5.57**[0.81]	5.33[0.82]	5.35[0.84]	**5.52**[0.93]	**5.65**[1.00]	**6.10**[1.27]
(D)						
MBNDMq	**20.31**$^{(8)}$[1.59]	**20.60**$^{(8)}$[1.76]	**22.19**$^{(8)}$[2.20]	**25.30**$^{(8)}$[3.07]	**31.03**$^{(8)}$[4.94]	**43.94**$^{(8)}$[8.98]
WMq	$21.06^{(8)}$[2.42]	$22.97^{(8)}$[4.26]	$27.33^{(8)}$[8.17]	$36.49^{(8)}$[15.30]	$55.09^{(8)}$[29.56]	$89.65^{(8)}$[58.70]
MPSSEF	29.78[7.72]	63.88[22.09]	103.30[52.07]	167.73[110.09]	291.27[227.61]	585.85[514.84]
MPSSEF^{32}	29.19[7.76]	30.42[7.91]	32.60[8.21]	38.46[9.34]	44.18[10.70]	52.12[13.18]

Table 3. Running times for 10 (A), 100 (B), 1.000 (C) and 10.000 (D) patterns on a protein sequence.

Backward DAWG Matching algorithm [23, 18], (WM) The Wu-Manber algorithm [20]. The MBNDM and WM algorithms have been run with different q-grams ranging from 3 to 8, and best times are reported with the regarding q.

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 [9] and are available online for download. We have generated sets of 10, 100, 1000 and 1000 patterns of fixed length ℓ for the tests. In all cases the patterns were randomly extracted from the text and the value ℓ was made ranging over the values 32, 64, 128, 256, 512, and 1024. For each case we reported the mean over the running times of 200 runs. The MPSSEF^q algorithm was tuned with q values 128, 64, 32 and 32 for searching sets of 10, 100, 1000, and 10000 patterns respectively. Tables 2, 3, and 4 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 means of the preprocessing and searching times. Best times have been boldfaced and underlined. Best searching times have been simply boldfaced. The q values presenting the length of the q–gram giving the best timing during the tests by WM and MBNDM algorithms are indicated as apices. The ESBOM algorithm is not included in the results since its running times are not competitive with the others.

Careful readers will notice that the MPSSEF algorithm gives better results than MPSSEF^q algorithm on sets of size 10 patterns, and might think that this contradicts with the motivation of MPSSEF^q. However, this is because the shift value in case of MPSSEF^q is $(q-16)$, where it is $(m/16)$ for MPSSEF. This fact becomes apparent when the number of patterns in the set is small. Though the MPSSEF^q uses less time in preprocessing as the number of patterns in the set is small, having a larger shift in MPSSEF dominates this advantage.

The size of the hash table F can be computed by the formula $64K \cdot 8 + (sublen - 15) \cdot (8 + 4 + 4)$ bytes, assuming an integer occupies 4 bytes and a pointer takes 8 bytes in practical situations. We have 64K pointers initially set to point empty lists at the beginning, and we need to reserve space for $(sublen-15)$ list nodes each of which has a next pointer, a pattern id, and the position on the regarding pattern. Notice that sublen is the length of the pattern in case of MPSSEF and q in case of MPSSEF^q ($q > 32$). In practice maximum memory requirement is measured to be less than 1MB on both.

On genome and natural language text the newly proposed algorithms outperforms in all cases the other solutions, which are known to be the fastest options. On protein sequences MPSSEF/MPSSEF^q performs better than the others up

(A) m	32	64	128	256	512	1024
MBNDMq	$4.26^{(5)}$[0.16]	$4.35^{(5)}$[0.17]	$4.58^{(5)}$[0.18]	$4.35^{(5)}$[0.17]	$4.35^{(5)}$[0.17]	$4.35^{(5)}$[0.17]
WMq	$4.11^{(6)}$[0.43]	$3.59^{(6)}$[0.43]	$3.20^{(6)}$[0.41]	$3.10^{(8)}$[0.42]	$2.83^{(8)}$[0.43]	$2.77^{(8)}$[0.48]
MPSSEF	**2.91**[0.01]	**2.39**[0.02]	**2.28**[0.06]	**2.16**[0.12]	**2.15**[0.23]	2.39[0.48]
MPSSEF^{128}	**2.91**[0.01]	**2.39**[0.02]	**2.28**[0.06]	2.38[0.06]	2.30[0.06]	**2.38**[0.06]
(B)						
MBNDMq	$7.65^{(8)}$[0.29]	$7.31^{(8)}$[0.27]	$7.39^{(8)}$[0.27]	$7.74^{(8)}$[0.30]	$7.63^{(8)}$[0.30]	$7.81^{(8)}$[0.32]
WMq	$5.28^{(8)}$[0.44]	$4.28^{(8)}$[0.45]	$3.92^{(8)}$[0.50]	$3.51^{(8)}$[0.56]	$3.67^{(8)}$[0.71]	$3.88^{(8)}$[0.99]
MPSSEF	**3.37**[0.08]	**2.98**[0.23]	**3.22**[0.53]	3.98[1.13]	5.09[2.27]	7.42[4.63]
MPSSEF^{64}	**3.37**[0.08]	**2.98**[0.23]	**3.22**[0.25]	**3.28**[0.26]	**3.32**[0.27]	**3.27**[0.29]
(C)						
MBNDMq	$16.87^{(5)}$[0.39]	$16.65^{(5)}$[0.40]	$17.34^{(5)}$[0.44]	$17.29^{(5)}$[0.49]	$18.06^{(5)}$[0.58]	$18.16^{(5)}$[0.82]
WMq	$13.35^{(8)}$[0.64]	$10.53^{(8)}$[0.80]	$9.66^{(8)}$[1.17]	$9.42^{(8)}$[1.85]	$10.52^{(8)}$[3.22]	$13.42^{(8)}$[6.05]
MPSSEF	**8.38**[0.80]	9.50[2.33]	12.79[5.28]	18.86[10.76]	33.77[22.06]	60.06[45.86]
MPSSEF^{32}	**8.38**[0.80]	**8.28**[0.84]	**8.10**[0.85]	**8.13**[0.89]	**8.54**[1.01]	**9.26**[1.38]
(D)						
MBNDMq	$118.2^{(5)}$[1.53]	$118.2^{(5)}$[1.70]	$120.8^{(5)}$[2.13]	$129.3^{(5)}$[3.28]	$136.7^{(5)}$[5.02]	$156.9^{(5)}$[8.92]
WMq	$114.6^{(5)}$[2.28]	$110.2^{(5)}$[4.06]	$108.4^{(5)}$[7.41]	$115.1^{(5)}$[14.05]	$129.6^{(5)}$[27.20]	$166.3^{(5)}$[54.04]
MPSSEF	**52.32**[7.64]	97.08[21.83]	162.1[51.76]	241.3[109.3]	376.3[224.0]	657.3[480.3]
MPSSEF^{32}	**52.32**[7.64]	**51.26**[7.75]	**56.44**[8.23]	**60.32**[8.99]	**70.13**[10.37]	**82.27**[12.84]

Table 4. Running times for 10 (A), 100 (B), 1.000 (C) and 10.000 (D) patterns on a natural language text.

to pattern set sizes of 10000. However, on larger sets MBNDM^q becomes faster. This is most probably due to fact that in the case of genome and natural language texts the shift heuristics of the MBNDM and WM algorithms lead to short shifts advancements on average because of the repetitive structure of the texts. It is important to note that on large sets of longer patterns the preprocessing time of MPSSEF algorithm dominates the searching time. Hence, for those cases, using the MPSSEF^q variant seems a better choice.

(A) m	32	64	128	256	512	1024
genome	1.57	1.50	1.47	1.46	1.39	1.23
protein	1.08	1.36	1.40	1.38	1.39	1.19
nat.lang.	1.41	1.50	1.40	1.43	1.31	1.16

(B) m	32	64	128	256	512	1024
genome	1.91	1.76	1.58	1.51	1.49	1.60
protein	1.31	1.35	1.21	1.16	1.18	1.20
nat.lang.	1.56	1.43	1.21	1.07	1.10	1.18

(C) m	32	64	128	256	512	1024
genome	4.42	4.64	4.78	4.65	4.58	4.35
protein	1.09	0.96	0.95	1.04	1.28	1.53
nat.lang.	1.59	1.27	1.19	1.15	1.23	1.44

(D) m	32	64	128	256	512	1024
genome	11.62	11.97	10.18	9.84	8.38	9.01
protein	0.69	0.67	0.68	0.65	0.70	0.84
nat.lang.	2.19	2.14	1.92	1.90	1.84	1.90

Table 5. The speed ups obtained via MPSSEF/MPSSEF^q algorithms during the experiments. Dividing the best timing achieved by MPSSEF or MPSSEF^q by the best of the competing algorithms (WM, MBNDM, and ESBOM) gives the ratios listed herein. Results on sets of 10 (A), 100 (B), 1.000 (C) and 10.000 (D) patterns.

(A) m	32	64	128	256	512	1024
genome	2.23	1.69	1.48	1.42	1.50	1.46
protein	2.05	1.60	1.24	1.21	1.32	1.35
nat.lang.	2.15	1.52	1.37	1.36	1.39	1.34

(B) m	32	64	128	256	512	1024
genome	3.43	2.87	2.90	2.89	2.89	2.97
protein	2.09	1.55	1.54	1.61	1.57	1.55
nat.lang.	2.88	2.24	2.22	2.19	2.20	2.28

(C) m	32	64	128	256	512	1024
genome	8.30	8.44	8.80	8.54	8.60	8.06
protein	2.19	2.18	2.22	2.19	2.18	2.11
nat.lang.	4.13	4.09	4.28	4.36	4.22	4.03

(D) m	32	64	128	256	512	1024
genome	19.05	19.12	16.91	16.54	14.94	15.90
protein	2.01	1.94	1.92	1.81	1.78	1.83
nat.lang.	7.55	7.43	7.21	7.26	7.19	7.38

Table 6. The speed ups obtained via MPSSEF^q algorithms compared with the non SSE implementation of the same algorithm. Results on sets of 10 (A), 100 (B), 1.000 (C) and 10.000 (D) patterns.

Table 5 summarizes the speed up ratios achieved via the new algorithms (larger the ratio, better the result). As can be viewed from that table, the newly proposed solutions are in general faster then the competitors in orders of magnitude. Notice that the gain in speed becomes more and more significant with the increasing size of the patterns sets.

To observe the gain we obtain by using the SSE instructions, we have also implemented the MPSSEF^q algorithm without using the SSE intrinsics. The comparisons of the SSE implementation versus non-SSE version is given in Table 6. The table shows that the gain obtained by vectorization decreases with the length of the patterns and increases with the size of the pattern sets. Hence, on larger sequences vectorization becomes more influential.

6 Conclusions

This study introduced a filter based algorithm for the multiple string matching problem, 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.

Considering the orders of magnitude performance gain reported with the experimental benchmarks, the presented algorithm becomes a strong alternative for multiple exact matching of long patterns The gain obtained in speed via MPSSEF 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, proposed algorithm is supposed to serve a good basis for massive multiple long pattern search applications on these areas.

The gain obtained via using the SSE technology might be more influential in parallel to the advancement of the single-instruction-multiple-data instructions. Although we have not tested MPSSEF on AVX technology, where there exists registers of size 256 bits, it is expected to have a similar speed-up on patterns larger than 64 symbols with the same algorithms.

References

1. A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.
2. R. Baeza-Yates and G.H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.
3. 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, 423–432, 2011.
4. D. Cantone, S. Faro, and E. Giaquinta. A Compact Representation of Nondeterministic (Suffix) Automata for the Bit-Parallel Approach. *Combinatorial Pattern Matching*, 288–298, 2010.
5. 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*, 11:25–36, 2012.
6. M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994.
7. S. Faro and T. Lecroq. Efficient variants of the backward-oracle-matching algorithm. *Int. J. Found. Comput. Sci.*, 20(6):967–984, 2009.
8. S. Faro and T. Lecroq. The exact string matching problem: a comprehensive experimental evaluation. *Arxiv preprint arXiv:1012.2547*, 2010.
9. S. Faro and T. Lecroq. Smart: a string matching algorithm research tool. Univ. of Catania and Univ. of Rouen, 2011. `http://www.dmi.unict.it/~faro/smart/`.
10. S. Faro and T. Lecroq. The exact online string matching problem: a review of the most recent results. *ACM Computing Surveys*, to appear.
11. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.
12. M.O. Külekci. Filter based fast matching of long patterns by using SIMD instructions. In *Proc. of the Prague Stringology Conference*, pages 118–128, 2009.
13. M.O. Külekci. Blim: A new bit-parallel pattern matching algorithm overcoming computer word size limitation. *Mathematics in Comp. Science*, 3(4):407–420, 2010.
14. G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *Comb. Pattern Matching*, pages 14–33. 1998.
15. G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM J. Experimental Algorithmics*, 5:4, 2000.
16. G. Navarro and M. Raffinot. *Flexible pattern matching in strings - practical on-line search algorithms for texts and biological sequences*. Cambridge Univ. Press, 2002.
17. Gonzalo Navarro and Kimmo Fredriksson. Average complexity of exact and approximate multiple string matching. *Theor. Comput. Sci.*, 321(2-3):283–290, 2004.
18. E. Rivals, L. Salmela, P. Kiiskinen, P. Kalsi, and J. Tarhio. Mpscan: Fast localisation of multiple reads in genomes. In *Proc. of WABI*, pages 246–260, 2009.
19. S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. of USENIX Winter 1992 Technical Conference*, pages 153–162,1992.
20. 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.
21. Sun Wu and Udi Manber. Fast text searching: allowing errors. *Commun. ACM*, 35(10):83–91, 1992.
22. S. Gog, K. Karhu, J. Krkkinen, V. Mkinen and N. Vlimki. Multi-Pattern Matching with Bidirectional Indexes. to appear in Proceedings of COCOON'12.
23. L. Salmela, J. Tarhio, J. Kyotojoki. Multi–pattern string matching with q–grams. *ACM J. Experimental Algorithmics*, 11, 2006.