Counting Mismatches with SIMD

Fernando J. Fiori*, Waltteri Pakalén, and Jorma Tarhio

Department of Computer Science
Aalto University, P.O.B. 15400
FI-00076 Aalto, Finland
fiorifj@gmail.com, waltteri.pakalen@aalto.fi, tarhio@iki.fi

Abstract. We consider the k mismatches version of approximate string matching for a single pattern and multiple patterns. For these problems we present new algorithms utilizing the SIMD (Single Instruction Multiple Data) instruction set extensions for patterns of up to 32 characters. We apply SIMD computation in two ways: in counting of mismatches and in calculation of fingerprints. We demonstrate the competitiveness of our solutions by practical experiments.

1 Introduction

The *string matching problem* is defined as follows: given a pattern $P = p_0 \cdots p_{m-1}$ and a text $T = t_0 \cdots t_{n-1}$ in an alphabet Σ, find all the occurrences of P in T. In this paper we consider the k *mismatches* variation of the problem where P' is an occurrence of P, if $|P'| = |P|$ holds and P' has at most k mismatches with P. The mismatch distance of two strings of equal length is also called the *Hamming distance*.

There are numerous good solutions for the k mismatches problem, see e.g. Navarro's survey [26]. In this article, we introduce new algorithms for the problem. Besides the single pattern problem, we also consider the multiple pattern variation. Our solutions utilize SIMD (Single Instruction Multiple Data) instruction set extensions [17,20]. We apply SIMD computation in two ways: in counting of mismatches and in calculation of fingerprints a.k.a. hash values. Our emphasis is on the practical efficiency of the algorithms and we show the competitiveness of the new algorithms by practical experiments. Our new algorithms for the single pattern problem are faster than reference methods in most cases tested, and our multiple pattern algorithm beats Fredriksson and Navarro's algorithm [13] with a wide margin.

The rest of the paper is organized as follows. Section 2 reviews earlier solutions. Section 3 introduces SIMD computation and the SIMD techniques applied. Section 4 and 5 describe the new algorithms, Section 6 presents the results of practical experiments, and Section 7 concludes the article.

2 Earlier Solutions

There are many algorithms for the string matching with k mismatches problem. Most of them solve the single pattern variation, whereas only few exist for the multiple pattern counterpart. Naively, an algorithm for the single pattern variation can be extended to solve the multiple pattern variation by executing it separately for each pattern. In the following, we will review earlier solutions to these variations.

* Visitor from Universidad Nacional de Rosario, Rosario, Santa Fe, Argentina.

Fernando J. Fiori, Waltteri Pakalén, Jorma Tarhio: Counting Mismatches with SIMD, pp. 51–61.
Proceedings of PSC 2017, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06193-0

2.1 Single String Matching with k Mismatches

A naive algorithm works in $O(mn)$ time in the worst case and in $O(kn)$ time on average if individual characters in P and T are chosen independently and uniformly from the alphabet Σ.

Baeza-Yates and Gonnet [3] presented Shift-Add (SA), the first bit-parallel algorithm for the k mismatches problem. Shift-Add works in linear time for short patterns $m \leq w/\lceil \log_2(k+1)+1 \rceil$ where w is the width of the computer word. Shift-Add is still competitive for short patterns and large k [15]. Ďurian et al. [9] presented variations of SA, TuSA and TwSA, which process the alignment window backwards.

Approximate Boyer–Moore (ABM) by Tarhio and Ukkonen [30] is a generalization of the Boyer–Moore–Horspool algorithm [18] to approximate string matching. In ABM, shifting is based on a q-gram, $q = k + 1$. Liu et al. [24] tuned ABM for small alphabets. Their algorithm applies wider q-grams and is called FAAST (short for a Fast Algorithm for Approximate STring matching). Salmela et al. [28] designed an enhanced version of FAAST. We call this algorithm EF. In Sect. 4.2 we will integrate SIMD computation with EF.

Approximate BNDM (ABNDM) by Navarro and Raffinot [27] is based on the BNDM algorithm [27] for exact string matching. BNDM simulates the suffix automaton of the reversed pattern with bit-parallelism. ABNDM as well as ABM, FAAST, EF, and TwSA achieve a sublinear running time on average in the case of favorable problem parameters.

The Baeza-Yates–Perlberg algorithm (BYP) [5] is based on a partitioning scheme, where at least one of P's substrings of length $l = \lfloor m/(k+1) \rfloor$ is exactly present in an approximate occurrence of P. In the preprocessing phase it splits the pattern into subpatterns of length l, and then it performs a multiple exact string matching search of these subpatterns. Whenever one of the subpatterns is found, it checks if there is an approximate pattern match with Ukkonen's dynamic algorithm [31]. In Sect. 4.3 we will integrate SIMD computation with BYP.

Besides practically oriented algorithms mentioned above, there are other solutions to the k mismatches problem: the kangaroo method [14,23], the algorithms based on the fast Fourier transform and marking [1,2,12], and the $O(nk^2 \log k/m + n$ polylog $m)$ solution presented by Clifford et al. [7], which is the best theoretical result.

2.2 Multiple String Matching with k Mismatches

The first algorithm for multiple string matching with k mismatches was presented by Muth and Manber [25]. For $k = 1$ they preprocess all the strings that result of taking one character out of every pattern, and compute a hash value for each of them, storing it in a table. This amounts to an $O(rm)$ preprocessing time for r patterns. For a text window of m characters, they compute m hash values in the same way as for the patterns. If there is a match in the hash table, a naive verification follows. The average search time is $O(mn(1 + rm^2/M))$, where M is the size of the hash table. If $M = \Omega(rm^2)$, this results in $O(mn)$. In this way, the total cost of the algorithm is $O(m(r + n))$. However, if $k > 1$ they have to preprocess all the strings that result from taking k characters out of every pattern, amounting for a total time complexity of $O(m^k(r + n))$. Hence, it allows only very small k in practice, but the overall complexity is rather independent of the number of patterns to search, given that n is usually much larger than r.

Later, Baeza-Yates and Navarro [4] designed an algorithm which is based on a similar partition scheme as in the BYP algorithm. They split every pattern into subpatterns of length $l = \lfloor m/(k+1) \rfloor$ and perform an exact multiple pattern search. Whenever there is a subpattern occurrence, they check for the entire pattern with an approximate single pattern matching algorithm.

The fastest algorithm to date is Fredriksson and Navarro's algorithm [13], which is optimal on average. It places a window over the text, in which q-grams are read in a backwards order. Whenever an occurrence is impossible, the window is shifted past the read q-grams. The average complexity of the algorithm is $O((k + \log_\sigma(rm))n/m)$ for $\alpha < 1/2 - O(1/\sqrt{\sigma})$, where $\alpha = k/m$ is the difference ratio.

3 SIMD Techniques

SIMD [20] is a type of parallel architecture that allows one instruction to be operated on multiple data items at the same time. Initially, SIMD was used in multimedia, especially in processing images or audio files. SIMD instructions have since found applications in other areas such as cryptography. Recently, they have also been applied to string matching [6,10,21,22,29].

Streaming SIMD Extensions comprise of SIMD instruction sets supported by modern processors which allow computation on vectors of length 16 bytes in the case of SSE2 and 32 bytes in the case of AVX2. In the near future, one can process 64 bytes with AVX-512. The instructions operate on such vectors stored in special registers. As one instruction is performed on all the data in these vectors, it is considered SIMD computation.

Next, we describe our techniques to use SIMD in the new algorithms. In the descriptions, the SSE2 instructions are listed for 16 bytes (= 128 bits). There are corresponding AVX instructions for 32 bytes (= 256 bits). We assume that a byte represents one character.

3.1 Counting of Mismatches

Counting mismatches is an usual operation in approximate string matching. It can be done with the instructions *simd-cmpeq*(x, y) and *simd-popcount*(x) explained below. In practice, we also need the instruction *simd-load*(x), which is an intrinsic function of the compiler formally defined as

$$\_\_m128i \ \_mm\_loadu\_si128(x).$$

This instruction loads 16 bytes from the address x to a SIMD register given as the left-hand side of an assignment statement. The instruction *simd-cmpeq*(x, y) is formally

$$\_mm\_movemask\_epi8(\_mm\_cmpeq\_epi8(\_\_m128i\ x, \_\_m128i\ y)).$$

The instruction $\_mm\_cmpeq\_epi8$ compares 16 bytes in x and y bytewise for equality and stores the result. The instruction $\_mm\_movemask\_epi8$ creates a bitvector from the most significant bit of each byte of the parameter. The instruction *simd-popcount*(x) counts the number of on bits in x and is formally

$$\_mm\_popcnt\_u32(x).$$

The *simd-cmpeq*(x, y) instruction, therefore, makes it possible to compare up to α characters at the same time, where α is 16 or 32. The result is a bitvector of the pairwise comparisons. Lastly, a popcount operation on the result tells the number of matching characters.

3.2 CRC as a Fingerprint

There are many filtration methods for approximate string matching. Those methods contain two phases which are usually interleaved. The filtration phase selects match candidates and the checking phase verifies them. The former often entails the calculation of a fingerprint or a hash value from a q-gram, with which precomputed tables are accessed. Such a calculation can be performed with the *simd-crc*(x) instruction. A similar instruction was first used by Faro and Külekci [10,11] in exact string matching.

The instruction *simd-crc*(x) returns a b-bit value by first calculating a 32-bit cyclic redundancy checksum (CRC) of a 64-bit value, and then taking the b least significant bits of the CRC. It is formally

$$\_mm\_crc32\_u64(x)\ \&\ mask,$$

where x is a 64-bit integer, $mask$ is $2^b - 1$, and '&' is bitparallel *and*. Based on our experiments, the best value of b depends on the problem parameters.

4 Improved Solutions – Single Pattern

4.1 Variations of Naive

A straightforward approach to string matching with k mismatches is the naive counting of mismatches. Alg. 1 is the pseudocode of the naive algorithm ANS (short for Approximate Naive with SIMD). ANS counts the character matches with P starting from the $n - m + 1$ first positions of the text. According to our experiments (see Section 6), it is clearly faster than both the classic Shift-Add [3] and TuSA [9].

Algorithm 1: ANS
$x \leftarrow$ *simd-load*$(p_0 \cdots p_{m-1})$
for $i \leftarrow 0$ to $n - m$ do
  $y \leftarrow$ *simd-load*$(t_i \cdots t_{i+\alpha-1})$
  $t \leftarrow$ *simd-cmpeq*(x, y)
  if *simd-popcount*$(t) \geq m - k$ then $occ \leftarrow occ + 1$

There is a way to make ANS even faster when α is 16. We preprocess the condition *simd-popcount*$(t) \geq m - k$ to a Boolean array D for each vector t of 16 bits. Then the last line of ANS is changed to

$$\text{if } D[t] \text{ then } occ \leftarrow occ + 1$$

We call this variation ANS2. ANS2 is about 30% faster than ANS in our experiments in Sect. 6.

For longer patterns, $16 < m \leq 32$, the last line will be

$$\text{if } D[t\ \&\ mask] \text{ then if } \textit{simd-popcount}(t) \geq m - k \text{ then } occ \leftarrow occ + 1$$

where $mask$ is $2^{16}-1$. In other words, the first 16 characters of the pattern are tested first.

In our test environment (see Sect. 6), the computation of D takes about 2 ms, which is tolerable. Note that the preprocessing time would grow exponentially if D were extended for wider vectors. The speed of ANS does not depend on k, while the speed of ANS2 obviously decreases when k approaches m for $m > 16$. Moreover, we observed a further decrease in practice, as discussed in Sect. 6.1.

Besides the *simd-cmpeq* instruction and other basic SIMD commands, the SIMD architecture comprises of several aggregation operations for string processing. However, they are too slow for the k mismatches problem on those processors we have tested. Hirvola [16] implemented several algorithms similar to ANS with PCMP and STTNI instructions, but all those algorithms are clearly slower than ANS and TuSA.

4.2 EF Enhanced with SIMD

EF contains a filtration and a checking phase. The checking method can be replaced with ANS2 (see Sect. 4.1), while the fingerprint computation of the filtration method can be replaced with the CRC fingerprint technique of Sect. 3. Alg. 2 shows the pseudocode of EF.

Algorithm 2: EF
$s \leftarrow m-1$
while $s < n$ do
  $f \leftarrow \sum_{i=0}^{q-1} map(t_{s-i}) * 4^i$
  if $M[f] \leq k$ then
    $c \leftarrow M[f]$
    for $i \leftarrow 1$ to $m-q$ do
      if $t_{s-q-i+1} \neq p_{m-q-i}$ then
        $c \leftarrow c+1$
        if $c > k$ then break
    if $c \leq k$ then $occ \leftarrow occ+1$
  $s \leftarrow s + S_q[f]$

For each q-gram $u_0 \cdots u_{q-1}$, the preprocessing phase of EF computes the Hamming distance with the end of all prefixes of the pattern. With this information, a shift table S_q can be constructed (see details in [28]). M is another precomputed table. M gives the Hamming distance of a q-gram against the last q-gram of the pattern. Whenever $M[t_{s-q+1} \cdots t_s] > k$ holds, the algorithm shifts forward without processing the alignment window further. Both the tables are accessed with the fingerprint $f \leftarrow \sum_{i=0}^{q-1} map(t_{s-i}) * 4^i$, where the function map maps each DNA character to an integer in $\{0,1,2,3\}$.

Alg. 3 is the pseudocode of EFS, which is EF enhanced with SIMD computation for $m \leq 16$. The array D is computed in the same way as for ANS2. For longer patterns, $16 < m \leq 32$, the required change is the same as in the case of ANS2.

4.3 BYP Enhanced with SIMD

BYP looks for exact occurrences of substrings of length $l = \lfloor m/(k+1) \rfloor$ (called subpatterns from now on) of the pattern in the text. To achieve this, we employed a tuned version of MEPSM algorithm [11] for exact multiple string matching. MEPSM

Algorithm 3: EFS
$x \leftarrow$ *simd-load*$(p_0 \cdots p_{m-1})$
$s \leftarrow m - 1$
while $s < n$ do
  $f \leftarrow$ *simd-crc*$(t_{s-q+1} \cdots t_s)$
  if $M[f] \leq k$ then
    $y \leftarrow$ *simd-load*$(t_{s-m+1} \cdots t_s)$
    $t \leftarrow$ *simd-cmpeq*(x, y)
    if $D[t]$ then $occ \leftarrow occ + 1$
  $s \leftarrow s + S_q[f]$

reports subpattern occurences, which are later verified by ANS2 (see Sect. 4.1). Let us call the total algorithm BYPS.

MEPSM computes the CRC fingerprint of every q-gram of each subpattern, where $q \leq l$ is a parameter of MEPSM. The information about which q-gram the fingerprint belongs to is stored in a table. Afterwards, during the search, the algorithm looks for matching fingerprints of q-grams in the text. Whenever a subpattern occurrence candidate is found, it is naively verified and reported in case of a match. After each q-gram analysis, the algorithm shifts forwards by $l - q + 1$ characters.

We tuned MEPSM by setting q as large as possible, which causes less fingerprint collisions. Conversely, larger q reduces shifts between alignments. However, this trade-off showed to be really satisfactory, especially in the case of small subpatterns.

Our algorithm has the practical limitation that $4 \leq l \leq 32$ must hold for l, as the performance drops substantially otherwise.

5 Improved Solution – Multiple Patterns

We have extended BYPS algorithm to work with multiple patterns. The new algorithm MBYPS works as follows:

1. In the preprocessing, we split every pattern into subpatterns of length l. Then we compute the CRC fingerprint of every q-gram of each subpattern, where $q \leq l$ is a parameter of the MEPSM algorithm. The fingerprint is used to access a table that stores information about which subpattern of which pattern it was computed from.
2. In the search, we compute the fingerprint of a q-gram in the text, with which we fetch the corresponding information from the table. We perform a shift of $l-q+1$ characters in the text after analysing each q-gram, which is the maximum number of characters we can skip.
3. For every subpattern associated with the fingerprint, we naively check if it appears exactly at this point. If it does, a possible occurrence of the corresponding pattern is reported.
4. Every time a match candidate of a pattern is found, we use an approximate single pattern matching algorithm to verify it.

For the phase of exact multiple string matching, we use the tuned version of MEPSM as in BYPS. For the phase of approximate single string matching, we use ANS2 for $m \leq 32$. For longer patterns, another method such as Ukkonen's dynamic algorithm [31] should be used.

The phase of approximate single string matching requires the occurrences of the subpatterns to be ordered, so as to avoid re-verifying an occurrence. However, MEPSM does not guarantee ordering. This has been solved by executing the approximate single pattern matching algorithm in a larger window. If a subpattern occurrence is found at position x in the text, we check for an approximate pattern occurrence from position $x-(m-l)$ to $x+m$. Thus, once an occurrence of a pattern has been found, a newer occurrence will never precede it positionally, as shown next.

Justification. Let x and y be the text positions of an old and a new q-gram occurrence respectively, which correspond to exact subpatterns occurrences. These subpatterns are placed in text positions s_x and s_y respectively, such that $x-(l-q) \leq s_x \leq x$ and $y-(l-q) \leq s_y \leq y$. Then the patterns which contain such subpatterns occur at positions p_x and p_y respectively, such that $s_x-(m-l) \leq p_x \leq s_x$ and $s_y-(m-l) \leq p_y \leq s_y$. But we perform our approximate pattern matching search from $s_x-(m-l)$ onwards. So we need to check that:

$$s_x-(m-l) \leq p_y$$

Which is valid if:

$$s_x-(m-l) \leq s_y-(m-l) \iff s_x \leq s_y$$

Which is true if:

$$x \leq y-(l-q) \tag{1}$$

It could happen that $x=y$ but MEPSM reports first the highest occurrences of a determined supattern (i.e. it preserves ordering of occurrences of q-grams for the same subpattern). So $x+(l-q+1) \leq y$ because we skip $l-q+1$ bytes after analysing a q-gram. Then (1) is true if:

$$x \leq x+(l-q+1) \iff q-1 \leq l$$

Which is true because q is the size of the q-grams of the subpatterns of length l.

6 Experiments

The tests were run on Intel Core i7-6500U 2.5 GHz with 16 GiB memory. This processor has SSE2 and AVX2, but not AVX-512. Programs were written in the C programming language and compiled with gcc 5.4.0 using -O3 optimization level. All the algorithms were implemented and tested in the testing framework of Hume and Sunday [19].

We used two texts: DNA (the genome of E. Coli, 4.6 MB) and English (the KJV Bible, 4.0 MB) for testing. The texts were taken from the Smart corpus[1]. Sets of patterns of various lengths were randomly taken from each text. In the case of single pattern matching, each set contains 200 patterns.

A word of warning. Our experimental results hold on the processor we used in our tests. It is possible that future processors will give different results, if the relative speed of instructions will change. In exact string matching we have encountered such a development several times. For example, in the case of English text, SBNDM2 [8] is 75% faster than ufast-rev-md2 [19] on our test processor for $m=5$, but the situation is almost reversed on a 20 years older processor Pentium 75: ufast-rev-md2 is 47% faster than SBNDM2!

[1] https://www.dmi.unict.it/~faro/smart/

6.1 Single String Matching with k Mismatches

The new algorithms were compared with the following earlier algorithms: SA [3], TuSA [9], TwSA [9], EF [28] and BYP [5]. According to tests by Hirvola [16], TwSA was the best for English data. According to tests by Salmela et al. [28], EF was the best for DNA data.

The results are shown in Table 1 with the best times bolded. We can observe that ANS2 is the best for several parameter combinations on both DNA and English. Meanwhile, EFS and BYPS are the best for some cases with small k on DNA, and TwSA is the best on English for some combinations when $k > 1$ and $m \geq 16$.

Like SA, ANS and ANS2 work for all possible values of k, and ANS does so at an almost constant speed independent of the value of k. On the contrary, TuSA and TwSA are limited to small values of k for long patterns. For example, they only work for $k < 4$ in the case of $m = 20$. Furthermore, the speed of TwSA degrades as k grows. EF, EFS, BYP and BYPS exhibit similar behavior, with k affecting their speed. Despite this, the growth of k can be tolerated given that m is large enough, i.e. when we have a large difference ratios. Some timings of BYPS have been omitted because it does not work for $l < 4$.

ANS2 is the best for small patterns across both texts with every value of k. As the pattern length increases, EFS overtakes ANS2 on DNA, and TwSA overtakes ANS2 on English up to a small value of k. Once k surpasses this value, ANS2 becomes the best again.

Lastly, in Sect. 4.1 it was stated that the speed of ANS2 obviously decreases when k approaches m for $m > 16$. Beyond that, however, we observed a peak in the speed of ANS2 for $m > 16$, as depicted in Figure 1. We tried two different compilers, and multiple compilation options, but the peak persisted. It is conjectured to be caused by branch mispredictions. Thus, ANS is the better choice over ANS2 for $k > 7$ on DNA, and $k > 11$ on English when $m > 16$.

BYPS has also been tested for longer patterns. According to our experiments and following the same line as stated by Baeza-Yates and Perlberg in [5], BYP and BYPS obtain their best results for high difference ratios.

Table 1. Search times (in seconds) of algorithms for approximate matching with k mismatches.

	$m = 8$			$m = 12$			$m = 16$			$m = 20$			
k	1	2	3	1	2	3	1	2	3	1	2	3	Σ
SA	1.99	2.00	1.99	1.99	1.99	2.01	2.02	1.99	1.99	1.99	1.99	1.99	DNA
TuSA	1.66	1.66	1.66	1.66	1.66	1.66	1.66	1.66	1.66	1.66	1.66	1.66	
TwSA	1.73	2.31	2.63	1.16	1.54	1.85	0.88	1.15	1.39	0.71	0.92	1.12	
ANS	1.13	1.13	1.13	1.13	1.13	1.13	1.13	1.13	1.13	1.13	1.13	1.13	
ANS2	**0.76**	**0.76**	**0.76**	0.76	**0.76**	**0.76**	0.83	**0.85**	**0.90**	0.82	0.83	**0.86**	
EF	1.70	2.46	4.27	1.08	1.51	2.55	0.81	1.14	1.94	0.66	0.95	1.71	
EFS	1.09	1.78	3.76	**0.71**	1.13	2.12	0.55	0.91	1.73	0.46	**0.79**	1.63	
BYP	4.95	-	-	4.54	7.20	-	4.15	6.28	9.58	4.15	6.02	8.05	
BYPS	1.56	-	-	1.48	1.79	-	**0.42**	1.53	2.21	**0.41**	1.57	1.62	
SA	1.74	1.74	1.74	1.74	1.74	1.74	1.74	1.74	1.74	1.74	1.74	1.73	English
TuSA	1.45	1.45	1.45	1.45	1.45	1.45	1.45	1.45	1.45	1.45	1.45	1.45	
TwSA	0.78	1.14	1.52	0.61	0.83	1.07	0.49	**0.65**	0.82	0.43	**0.54**	**0.67**	
ANS	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	1.01	0.99	
ANS2	0.72	**0.66**	**0.67**	0.66	**0.66**	**0.66**	0.66	0.66	**0.66**	0.72	0.72	0.72	
BYP	1.39	-	-	1.03	1.83	-	0.86	1.50	2.16	0.72	1.27	1.83	
BYPS	**0.65**	-	-	**0.43**	0.78	-	**0.32**	0.82	0.96	**0.31**	0.55	0.87	

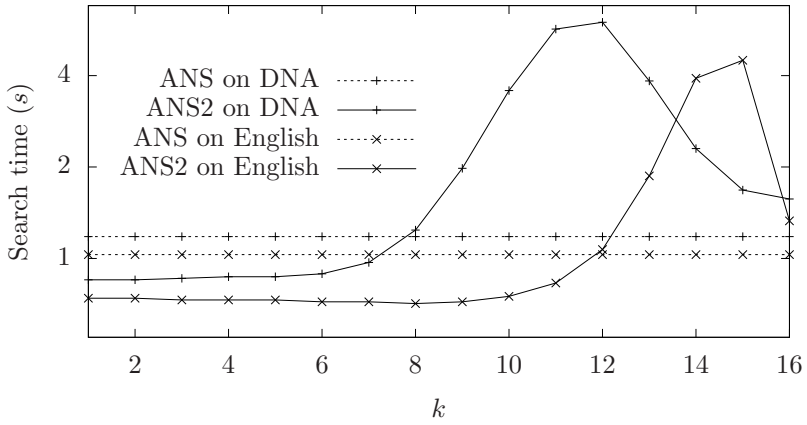

Figure 1. Search times of ANS and ANS2 as a function of k for $m = 20$.

6.2 Multiple String Matching with k Mismatches

We compare our new MBYPS algorithm with Fredriksson and Navarro's algorithm (FN) [13]. We used sets of 10, 100 and 1000 patterns. The results show that our algorithm outperforms FN in all cases. There is a larger difference for larger sets of patterns, and for larger difference ratios. We also ran tests on a Protein sequence and obtained similar results.

It is worth mentioning that we performed thorough testing to choose the best parameters for FN in each case. For DNA we obtained the same tuning mentioned in [13] as the best configuration.

Table 2. Search times (in seconds) for multiple approximate matching with k mismatches.

	m=8	$m = 16$			$m = 24$			$m = 32$				
k	1	1	2	3	1	2	3	1	2	3	r	Σ
FN	0.129	0.018	0.120	0.396	0.006	0.009	0.015	0.004	0.006	0.008	10	DNA
MBYPS	0.031	0.007	0.019	0.066	0.002	0.008	0.013	0.001	0.003	0.008		
FN	1.132	0.165	0.996	4.635	0.012	0.032	0.086	0.007	0.016	0.034	100	
MBYPS	0.240	0.010	0.154	0.585	0.003	0.012	0.069	0.002	0.005	0.013		
FN	11.220	1.697	10.222	44.030	0.098	0.364	1.044	0.059	0.158	0.344	1000	
MBYPS	2.574	0.033	1.695	6.797	0.012	0.055	0.695	0.011	0.025	0.075		
FN	0.026	0.008	0.014	0.027	0.005	0.008	0.012	0.004	0.006	0.009	10	English
MBYPS	0.013	0.006	0.008	0.008	0.001	0.006	0.007	0.001	0.002	0.006		
FN	0.143	0.043	0.173	0.406	0.023	0.104	0.174	0.021	0.084	0.125	100	
MBYPS	0.046	0.009	0.048	0.093	0.002	0.009	0.021	0.001	0.003	0.010		
FN	1.723	0.373	1.212	4.224	0.201	0.491	1.028	0.148	0.333	0.563	1000	
MBYPS	0.314	0.022	0.348	0.984	0.008	0.032	0.167	0.007	0.018	0.045		

7 Concluding Remarks

We have demonstrated that simple SIMD solutions are competitive in searching for approximate single pattern matches within the Hamming distance for patterns $|P| \leq 32$. In Sect. 4.1 and Sect. 4.2, we showed that the algorithms for naive counting of mismatches can be used as a checking method for single pattern filtration algorithms.

Meanwhile, the fingerprint calculation of a filtration method can be replaced with the CRC fingerprint technique of Sect. 3.2.

We have also presented an effective way of using the SIMD techniques for approximate multiple string matching in Sect. 5. The resulting algorithm is substantially faster than the previous most competitive algorithm across multiple alphabets.

When AVX-512 will become widely available, it may be possible to achieve better speed-ups, because compare and mask instructions have been merged.

References

1. K. Abrahamson: *Generalized string matching*. SIAM Journal on Computing, 16(6) 1987, pp. 1039–1051.
2. A. Amir, M. Lewenstein, and E. Porat: *Faster algorithms for string matching with k mismatches*. Journal of Algorithms, 50(2) 2004, pp. 257–275.
3. R. Baeza-Yates and G. H. Gonnet: *A new approach to text searching*. Communications of the ACM, 35(10) 1992, pp. 74–82.
4. R. Baeza-Yates and G. Navarro: *New and faster filters for multiple approximate string matching*. Random Structures & Algorithms, 20(1) 2002, pp. 23–49.
5. R. Baeza-Yates and C. Perlberg: *Fast and practical approximate string matching*. Information Processing Letters, 59(1) 1996, pp. 21–27.
6. T. Chhabra, S. Faro, M. O. Külekci, and J. Tarhio: *Engineering order-preserving pattern matching with SIMD parallelism*. Software: Practice and Experience, 47(5) 2017, pp. 731–739.
7. R. Clifford, A. Fontaine, E. Porat, B. Sach, and T. Starikovskaya: *The k-mismatch problem revisited*, in Proc. 27th ACM-SIAM Symposium on Discrete Algorithms, SIAM, 2016, pp. 2039–2052
8. B. Ďurian, J. Holub, H. Peltola, and J. Tarhio: *Improving practical exact string matching*. Information Processing Letters, 110(4) 2010, pp. 148–152.
9. B. Ďurian, T. Chhabra, S. Guman, T. Hirvola, H. Peltola, and J. Tarhio: *Improved two-way bit-parallel search*, in Proceedings of the Prague Stringology Conference, Prague, Czech Republic, 2014, pp. 71–83
10. S. Faro and M. O. Külekci: *Fast packed string matching for short patterns*, in Proc. 15th Meeting on Algorithm Engineering and Experiments, SIAM, 2013, pp. 113–121.
11. S. Faro and M. O. Külekci: *Towards a very fast multiple string matching algorithm for short patterns*, in Proceedings of the Prague Stringology Conference, Prague, Czech Republic, 2013, pp. 78–91.
12. K. Fredriksson and S. Grabowski: *Exploiting word-level parallelism for fast convolutions and their applications in approximate string matching*. European Journal of Combinatorics, 34(1) 2013, pp. 38–51.
13. K. Fredriksson and G. Navarro: *Average-optimal single and multiple approximate string matching*. ACM Journal of Experimental Algorithmics, 9 2004.
14. Z. Galil and R. Giancarlo: *Improved string matching with k mismatches*. ACM SIGACT News, 17(4) 1986, pp. 52–54.
15. S. Grabowski and K. Fredriksson: *Bit-parallel string matching under Hamming distance in $O(n\lceil m/w\rceil)$ worst case time*. Information Processing Letters, 105(5) 2008, pp. 182–187.
16. T. Hirvola: *Bit-parallel approximate string matching under Hamming distance*. Master's Thesis, Aalto University, 2016. http://urn.fi/URN:NBN:fi:aalto-201608263081
17. M. Hassaballah, S. Omran, and Y. B. Mahdy: *A review of SIMD multimedia extensions and their usage in scientific and engineering applications*. The Computer Journal, 51(6) 2008, pp. 630–649.
18. R. N. Horspool: *Practical fast searching in strings*. Software: Practice and Experience, 10(6) 1980, pp. 501–506.
19. A. Hume and D. Sunday: *Fast string searching*. Software: Practice and Experience, 21(11) 1991, pp. 1221–1248.
20. Intel: *Intel (R) 64 and IA-32 architectures software developer's manual*. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (Retrieved in May 2017).

21. M. O. Külekci: *Filter based fast matching of long patterns by using SIMD instructions*, in Proceedings of the Prague Stringology Conference, Prague, Czech Republic, 2009, pp. 118–128.
22. S. Ladra, O. Pedreira, J. Duato, and N. R. Brisaboa: *Exploiting SIMD instructions in current processors to improve classical string algorithms*, in Proc. 16th East European Conference on Advances in Databases and Information Systems, LNCS, vol. 7503, Springer, 2012, pp. 254–267.
23. G. Landau and U. Vishkin: *Efficient string matching with k mismatches*. Theoretical Computer Science, 43 1986, pp. 239–249.
24. Z. Liu, X. Chen, J. Borneman, and T. Jiang: *A fast algorithm for approximate string matching on gene sequences*, in Proc. 16th Symposium on Combinatorial Pattern Matching, LNCS, vol. 3537, Springer, Berlin, 2005, pp. 79–90.
25. R. Muth and U. Manber: *Approximate multiple string search*, in Proc. 7th Symposium on Combinatorial Pattern Matching, LNCS, vol. 1075, Springer, Berlin, 1996, pp. 75–86.
26. G. Navarro: *A guided tour to approximate string matching*. ACM Computing Surveys, 33(1) 2001, pp. 31–88.
27. G. Navarro and M. Raffinot: *Fast and flexible string matching by combining bit-parallelism and suffix automata*. Journal of Experimental Algorithmics, 5 2000, pp. 4.
28. L. Salmela, J. Tarhio, and P. Kalsi: *Approximate Boyer-Moore string matching for small alphabets*. Algorithmica, 58(3) 2010, pp. 591–609.
29. J. Tarhio, J. Holub, and E. Giaquinta: *Technology beats algorithms (in exact string matching)*. To appear in: Software: Practice and Experience.
30. J. Tarhio and E. Ukkonen: *Approximate Boyer-Moore string matching*. SIAM Journal on Computing, 22(2) 1993, pp. 243–260.
31. E. Ukkonen: *Finding approximate patterns in strings*. Journal of Algorithms, 6(1) 1985, pp. 132–137.