Jumbled Matching with SIMD

Sukhpal Singh Ghuman and Jorma Tarhio

Department of Computer Science
Aalto University
P.O. Box 15400, FI-00076 Aalto, Finland
firstname.lastname@aalto.fi

Abstract. Jumbled pattern matching addresses the problem of finding all permuted occurrences of a substring in a text. We introduce two improved algorithms for exact jumbled matching of short patterns. Our solutions apply SIMD (Single Instruction Multiple Data) computation in order to quickly filter the text. One of them utilizes the equal any operation and the other searches for the least frequent character of the pattern. Our experiments show that the best algorithm is 30% faster than previous algorithms for short English patterns.

1 Introduction

Given a text $T = t_0 t_1 \cdots t_{n-1}$ and pattern $P = p_0 p_1 \cdots p_{m-1}$ over a finite alphabet Σ of size σ, the task of exact string matching [30] is to find all the occurrences of P in T, i.e. all the positions i such that $t_i t_{i+1} \cdots t_{i+m-1} = p_0 p_1 \cdots p_{m-1}$. In jumbled pattern matching [7,10], the aim is to find all substrings of T which are permutations of P. Jumbled matching is also known as permutation matching or Abelian matching. In other words, a substring u of T is a jumbled equivalent to P if the count of each character in P is equal to its count in u and $|P| = |u|$ holds. For example, a permutation *abecd* of the pattern $P = edcba$ occurs in the text $T = a\underline{abecd}cddee$.

Parikh vectors [33] can be used to identify jumbled substrings. Over a finite ordered alphabet, a Parikh vector $p(S)$ is defined as the vector of multiplicities of the characters of a string S. For instance, if $S = baadabbdd$ be a string over $\Sigma = \{a, b, c, d\}$, then the Parikh vector $p(S)$ is (3,3,0,3).

Initially, simple counting solutions [17,25,29] have been presented for jumbled pattern matching. These solutions work in linear time. The main idea of these algorithms is to scan the text from left to right and maintain counts of characters in a sliding alignment window of text. Originally, these counting algorithms were developed as filtration methods for online approximate string matching, but they recognize jumbled patterns as a side-effect when no errors are allowed. Also many other algorithms [6,8,13,16] have been introduced for jumbled pattern matching.

In this article, we introduce new algorithms for exact jumbled matching for short patterns. Our solutions apply the SIMD (Single Instruction Multiple Data) extensions of the SSE technology [18,22,24] which makes possible to process multiple characters at the same time. In this way, we are able to process the text in chunks of 16 characters resulting in faster execution for short patterns. We present two new algorithms based on filtration with SIMD instructions. One of them applies the equal any SIMD operation and the other searches for the least frequent character of the pattern. Our emphasis is on the practical efficiency of the algorithms and we show the competitiveness of the new algorithms by practical experiments. The best one of our algorithms achieves a speed-up of 30% for short English patterns.

Sukhpal Singh Ghuman, Jorma Tarhio: Jumbled Matching with SIMD, pp. 114–124.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8

The bit operations represented in pseudocodes are identical to the notations used in the C programming language. The operator $<<$ represents the left shift operation and $>>$ corresponds to the right shift operation.

The rest of the paper is organized as follows. Section 2 contains an extensive review on applications of jumbled pattern matching, Section 3 is an introduction to SIMD computation, Section 4 reviews earlier solutions, Sections 5 and 6 describe our new solutions, Section 7 presents the results of practical experiments, and Section 8 concludes the article.

2 Applications of Jumbled Matching

In recent past, several variations of jumbled pattern matching have originated. The problem has numerous applications in the field of bioinformatics such as alignment of strings [1], SNP discovery [3], discovery of repeated patterns [14], and the interpretation of mass spectrometry data [2]. Other applications [5] of jumbled pattern matching include string matching with a dyslectic word processor, table rearrangements, anagram checking, scrabble playing, and episode matching.

2.1 Gene Clustering

Jumbled matching can be used to find those genes that are closely related to one another. Sequencing of genome has become a regular practice in the last few decades, which in turn has led to the analysis of genomes at gene level [11,36] and the correlation of genes. Genes having similar functionality are correlated to each other. Gene clustering [31,26] helps to find the genes that are closely related to each other irrespective of the order in which they occur. Consistent occurrences of genes in the close proximity across genomes are believed to be functionally related. However, the order of the genes in chromosomes may be different. These kind of gene clusters help in solving the problem of local alignment of genes.

In the case of discovery of repeated patterns [14], jumbled matching algorithms can be used to solve the problem of local alignment of genes. However, the order of the occurrences of genes may not be the same. These can be usually modeled by Parikh fingerprints or character sets [4]. In other words, a group of genes that can appear in different order in genomes may have similarity.

2.2 Composition Alignment

In composition matching [1], the main idea is to match among the substrings that have similar composition or order. Composition alignment of two strings P and T having a scoring function $SF(p, t)$ is defined as the composition match between the substrings p and t of P and T as well as the single character match between P and T. The task is to find the best scoring alignment.

For example, let $P = GACTGTTATTCCTA$ and $T = GCATGTGGGGATCC$ be two strings over the alphabet $\Sigma = (A, C, G, T)$. One possible composition alignment for P and T is:

$$\underline{GAC}\mathbf{TGT}TATT\underline{CCTA}$$
$$\underline{CGA}\mathbf{TGT}GGGG\underline{ATCC}$$

Here, the characters in bold are used to depict exact single character matches and substring composition matches are represented by underlined substrings. Note that composition matches can occur consecutively in an alignment.

Composition alignment is one of the major applications of jumbled pattern matching. In standard alignment of two strings, each character of one string is matched against every single character of other string. However, in composition alignment matching the substrings that have the same characters are matched against the substring of the other string, even though the order of characters in the string can be different. It is easy to identify the subsequences that contain substrings of similar compositions. In other words, composition alignment is referred as pairing of substrings of exactly matching composition separated by insertions, deletions, or mismatches.

2.3 Mass Spectrometery and SNP Discovery

Jumbled pattern matching can also be used in the field of interpretation of mass spectrometer [2]. It is used to find the strings which have the same spectra. Mass spectra are simulated for every potential sequence and the resulting simulated spectra are then compared against the measured mass spectrum.

A single nucleotide polymorphism (SNP) is a variation at a single position in a DNA sequence among individuals. Each SNP represents a difference of a nucleotide in a single DNA. SNPs occur normally throughout a person's DNA. SNPs are believed to contribute strongly to the genetic variability in living beings. A comparatively new method to discover such polymorphisms is based on base-specific cleavage, where resulting cleavage products are analyzed by mass spectrometry. Simulating the mass spectrum that results from a base-specific cleavage experiment is relatively simple [3] and can be compared with simulating the mass spectrum of a protein.

3 SIMD

SIMD [22] is a type of parallel architecture that allows one instruction to be operated at the same time on multiple data items. Initially, SIMD has been used in multimedia especially in processing images or audio files. SIMD instructions have also found applications in many other areas like crytography. A detailed review of SIMD and its applications is given in [18,20]. Recently, SIMD instructions have also been applied to string matching [27,28].

SSE (Streaming SIMD Extensions) [24] comprise of SIMD instruction sets supported by modern processors which are capable of parallel execution of operations on multiple data simultaneously through a set of special instructions that work on limited number of special registers. SIMD instructions use sixteen 128-bit registers known as XMM0, XMM1, ..., XMM15. In our algorithms, we use specialized string matching SIMD instructions in addition to standard SIMD instructions.

The SIMD architecture comprises of several aggregation operations that can be applied on strings to process them simultaneously. Some of the aggregation operations that can be used in string processing are equal each, equal any, and ranges. In our approach, we have applied the equal any operation to speed up reading the text. The operation has two input operands which are strings of up to 16 characters. The first string represents a multiset of characters. The second string is the text itself. The output of the operation is a bitvector of 16 bits, where 1 means that the corresponding character in the string belongs to the set and 0 means the opposite.

For instance, let us consider a string *abbc* representing the multiset $\{a, b, b, c\}$ and a string *adcdbeaeebefdbce*. The output of the equal any operation is 1010101001000110 in the reverse order.

We have used the following SIMD instructions in our algorithms. The instruction *simd-load* is formally

$$\_\_m128i\_mm\_loadu\_si128(\_\_m128i\ const*\ mem\_addr).$$

This load instruction for SSE memory operations loads 16 bytes from the address to an SIMD register of the memory location mentioned as a parameter. The instruction, *simd-equal-any*(x, y) is formally

$$\_mm\_extract\_epi16(\_mm\_cmpistrm(x, y, \text{\_SIDD\_CMP\_EQUAL\_ANY}), 0).$$

The inner instruction has three parameters. The first and second parameters are string fragments with a maximum size of 16 bytes. The third parameter is a constant determining the type of comparison to be performed and the format of value to be returned. In our case, the third parameter is _SIDD_CMP_EQUAL_ANY. The instruction *_mm_extract_epi*16 extracts a selected signed or unsigned 16-bit integer from the first of its parameters.

The instruction *simd-cmpeq*(x, y) is formally

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

The instruction *_mm_movemask_epi*8$(__m128i\ z)$ creates a mask from the most significant bit of each 8-bit element in the parameter z and stores the result. The instruction $\_\_m128i\_mm\_cmpeq\_epi8(\_\_m128i\ x, \_\_m128i\ y)$ compares packed 8-bit integers in x and y bytewise for equality and stores the result.

The performance of SIMD instructions depends on the architecture of the processor. The performance of a single instruction is measured by latency and throughput. Latency is the number of cycles taken by the processor to give the desired outcome form the given input. Throughput refers to the number of cycles between subsequent calls of the same instruction. We used processors Intel i7-860 and i5-4250U in our experiments. Their microarchitectures are Nehalem and Haswell [21], respectively. The latency and throughput of the SIMD instructions used in our algorithms for these processors are given in Table 1. One should observe that string matching instructions are slower than ordinary SIMD instructions. For other processors the difference may be still larger. Therefore it is crucial which SIMD instructions an algorithm designer selects for his code.

Architecture	SIMD instruction	Latency	Throughput
Nehalem	*_mm_cmpistrm*	8	2
	*_mm_extract_epi*16	3	1
	*_mm_cmpeq_epi*8	1	0.5
	*_mm_movemask_epi*8	1	1
Haswell	*_mm_cmpistrm*	11	3
	*_mm_extract_epi*16	3	1
	*_mm_cmpeq_epi*8	1	0.5
	*_mm_movemask_epi*8	3	1

Table 1. Latency and throughput of SIMD instructions for Nehalem and Haswelll [23].

4 Earlier Solutions to Jumbled Matching

In this section, we present some earlier solutions for jumbled pattern matching. In recent past, many algorithms have been presented in this area. Grossi & Luccio's and Navarro's solutions [17,25,29] are based on the counts of characters occurring in the pattern and in an alignment window. These methods solve this problem in linear time. Navarro's counting algorithm is based on a sliding window approach.

Ejaz [13] proposed several algorithms for jumbled pattern matching. One of them utilizes backward scanning of the alignment window. Moreover, Burcsi et al. [6] introduced a light indexing approach with linear construction time and with sublinear expected query time.

According to the tests by Chhabra et al. [9], BAM, BAM2, and EBL are the fastest algorithms of jumbled matching for short English patterns. We use them as reference methods. Below we explain their main ideas.

BAM. The BAM (Bit-parallel Abelian Matcher) algorithm was presented by Cantone and Faro [8]. The algorithm applies backward scanning of the alignment window using bit-parallelism. The central idea of this algorithm is to assign a counter to each distinct character of the pattern and to allocate a bit field of a word for each counter. In addition to that, a common one bit counter is reserved for the remaining characters of the alphabet which do not occur in the pattern.

BAM2. Chhabra et al. [9] presented BAM2. BAM2 is a variation of BAM that handles a 2-gram at a time. It processes the whole alignment window with 2-grams (except the leftmost character in the case of odd m). This is beneficial because the alignment window is scanned on average further to the left in jumbled matching than in ordinary string matching. Moreover, 2-grams instead of single characters are read in our implementation of BAM2.

EBL. EBL (Exact Backward for Large alphabets) presented by Chhabra et al. [9] is based on the SBNDM2 algorithm [12]. EBL works on a sliding window approach. Characters in the window are read from right to left. EBL shifts the text window from left towards right whenever a mismatch occurs. SBNDM2 is a sublinear bit-parallel algorithm for exact string matching. In EBL, an array B corresponding to the incidence vector of SBNDM2 states if the character c is present in the pattern. $B[c]$ is assigned value 1 if c is present, otherwise $B[c]$ is 0. As in SBNDM2, two characters are read before the first test in an alignment window.

5 Equal Any Approach

In Sections 5 and 6 we present two new algorithms. Both solutions apply SIMD instructions in order to filter out a significant portion of text. The first algorithm utilizes the equal any SIMD command.

Let us assume that $m < 16$ holds. The width of a test window in the text is 16. The equal any SIMD command returns a bitvector k of 16 bits showing the positions in the test window which hold any character of the pattern. For example, if the test window is *this is a sample* and the pattern *aeiou*, the vector is:

```
elpmas a si siht
1000100100100100
```

Note that the orientation of the bitvector is the opposite of the text. A match candidate is found if the last m bits of the vector are ones. Note that such a case is only a match candidate because the counts of characters are not analyzed. For example, the string *aaaaa* is a match candidate for *abcde*.

Algorithm 1 EA$(P = p_0p_1 \cdots p_{m-1}, T = t_0t_1 \cdots t_{n-1})$

```
 1: Place a copy of P after T
 2: Call PP(m)
 3: occ ← 0; x ← simd-load(P)
 4: shift ← 1; i ← 0
 5: while true do
 6:     while shift > 0 do
 7:         y ← simd-load(t_i ··· t_{i+15})
 8:         k ← simd-equal-any(x, y)
 9:         shift ← d[k]
10:         i ← i + shift
11:     occ ← occ + verify(t_i ··· t_{i+m-1})
12:     if i = n then
13:         return occ − 1
14:     i ← i + 1
```

Alg. 1 is the pseudocode of the scanning algorithm EA based on the equal any approach. In EA, we use a table d for shifting the test window. The algorithm applies a skip loop with a stopper, which is a copy of the pattern. A heuristic algorithm called PP to compute d from m is given as Alg. 2. When a block of m ones is found in EA, the window is shifted so that the block is at the right end of the bitvector corresponding to the test window. When a block of m ones is at the right end, a match candidate is found. The entry of d is zero in such a case in order to get out from the skip loop.

Let us study more details of EA. The SIMD register x holds the pattern and the SIMD register y holds a test window of 16 bytes of the text. The registers x and y are processed with the *simd-equal-any* operation (see Sect. 3) resulting a 16-bit integer k on line 8. If $d[k] = 0$ holds, a match candidate is found and the inner loop is exited. On line 11 there is a call of a verification routine which verifies the match candidate. Any previous algorithm for jumbled pattern matching (especially BAM, BAM2, or EBL) can be used as a verification method. The variable *occ* holds the count of matches. The stopper creates a superfluous match which is subtracted from the count on line 13.

Let us consider how the shift table d is computed in PP. The table d is indexed with the 16-bit integer k. The computation consists of several subsequent for-loops which are in the decreasing order of shift. This order of computation is essential, because a single entry of d may be assigned several times.

Algorithm 2 PP(m)

```
 1: L ← 2^16 − 1; b ← 1 << 15
 2: for i ← 0 to b − 1 do
 3:       d[i] ← 16
 4: a ← b; b ← 3 << 14
 5: for x ← 15 downto 16 − m do
 6:       for i ← a to b − 1 do
 7:             d[i] ← x
 8:       a ← b; b ← b + (1 << (x − 2))
 9: s ← (1 << m) − 1; a ← s << (15 − m)
10: b ← 1 << 15; c ← 1 << (14 − m)
11: for x ← 15 − m downto 2 do
12:       for i ← a step b to L do
13:             for j ← 0 to c − 1 do
14:                   d[i + j] ← x
15:       a ← a >> 1; b ← b >> 1; c ← c >> 1
16: for i ← a step b to L do
17:       d[i] ← 1
18: for i ← s step s + 1 to L do
19:       d[i] ← 0
```

Let us go through the phases of Alg. PP in detail. Let the 16 bits of k be named by $k_{15}, k_{14}, \ldots, k_0$. In the beginning, the integer b corresponds to a bitvector of one followed by 15 zeros. When the leftmost bit k_{15} is zero, the test window can be shifted 16 positions in the best case (lines 1–3).

On lines 4–8 we consider the case where k starts with $16 - x$ ones followed by a zero for $x = 15, 14, \ldots, 16 - m$. Then the shift is x. For example, the shift for 1111010101001100 is 12 for $m > 4$.

On lines 9–15 we consider the case where k holds a block of m ones starting from $k_{14}, k_{13}, \ldots$ or k_{m+1}. The loop for x traverses all the possible locations of the block of m ones. The loop for i traverses all bit combinations of the first $16 - x - m$ bits. The loop for j traverses all bit combinations of the last $x - 1$ bits. For example, in the computation of $d[0011111101000100]$ for $m = 6$, a is 0011111100000000, b is 0100000000000000, c is 0000000010000000, and x is 8.

When the block of m ones ends at k_1, the shift is one and it is computed in a single loop (lines 16–17). When the block of m ones ends at k_0, a match candidate is found, and this is expressed as assigning a zero to all such entries (lines 18–19)

6 Least Frequent Character Approach

Our second approach was developed for natural language. We use SIMD instructions to analyze whether a test window of 16 bytes holds the least frequent character of the pattern. The frequency of characters is based on the text or on the language.

Alg. 3 is the pseudocode of the scanning algorithm LF based on the least frequent character approach. On line 11 there is a call of a search routine which searches a block of up to $2m + 14$ characters. Any previous algorithm for jumbled pattern matching (especially BAM, BAM2, or EBL) can be used as a search method. The parameter R is an array containing 16 bytes, each of which holding the least frequent character. The SIMD register x holds R and the SIMD register y holds a test window of 16 bytes of the text. The registers x and y are compared by the *simd-cmpeq* operation (see Sect. 3) on line 6. The algorithm applies a skip loop with a stopper, which is a

Algorithm 3 LF$(P = p_0p_1 \cdots p_{m-1}, T = t_0t_1 \cdots t_{n-1}, R)$

```
 1: Place a copy of P after T
 2: occ ← 0; i ← 0; f ← 0
 3: x ← simd-load(R)
 4: while true do
 5:     y ← simd-load(t_i ··· t_{i+15})
 6:     while simd-cmpeq(x, y) = 0 do
 7:         i ← i + 16
 8:         y ← simd-load(t_i ··· t_{i+15})
 9:     if f < i − m + 1 then
10:         f ← i − m + 1
11:     occ = occ + search(t_f ··· t_{i+15+m−1})
12:     f ← i + 16
13:     if f ≥ n then
14:         return occ − 1
15:     else
16:         i ← i + 16
```

copy of the pattern. As in Alg. EA, the stopper creates a superfluous match which is subtracted from the count on line 14.

We use an additional variable f to control the starting position of a block. If the previous block has been skipped then the leftmost possible starting position for a match is $i + m - 1$. Otherwise the leftmost possible starting position for a match is i. Without such control, we would get a wrong number of matches, because there could be matches which would belong to two blocks.

7 Experiments

The tests were run on Intel 2.70 GHz i7-860 Nehalem processor with 16 GB of memory. All the algorithms were implemented in the C programming language and run in the 64-bit mode in the testing framework of Hume and Sunday [19]. We used two types of data for testing the algorithms. The protein text is 3 MB long and English text (KJV Bible) is 4 MB long. Both the texts were taken from the Smart corpus [15]. From both the texts, we picked six sets of 200 patterns with lengths $m = 4, 5, \ldots, 10$.

We tested the EA and LF algorithm schemes with BAM, BAM2, and EBL as the checking subroutine, i.e. six new algorithms. From these we selected LF-BAM2, EA-BAM2, and EA-EBL for further consideration. BAM, BAM2, and EBL were our reference methods. The running time of the PP algorithm for the EA scheme was about 10 ms.

In our tests we applied BAM2 without the bin sharing technique [9]. For patterns having at most 10 characters we do not require to use shared bins in the 64-bit architecture. One bit is reserved for characters which are not present in the pattern and five bits are enough for each bin.

Tables 2 and 3 present the average execution times in seconds for English and protein data, respectively. The results were retrieved as an average of ninety nine runs. The best execution times are highlighted by placing them in a box.

In Table 2 the execution time of the best one of the new algorithms is 17–33 percent less than the best time for earlier algorithms for $4 \leq m \leq 9$. EA-BAM2 is fastest for short patterns of length 4 and 5. LF-BAM2 performs best for the remaining pattern lengths except 10, where BAM2 has best execution time.

m	BAM	BAM2	EBL	LF-BAM2	EA-BAM2	EA-EBL
4	1.1918	1.4221	0.7274	0.6119	0.4995	0.5344
5	1.1942	0.7561	0.7364	0.5732	0.4903	0.5603
6	1.1037	0.5473	0.6891	0.4515	0.4859	0.5321
7	1.0116	0.4207	0.6502	0.3611	0.4809	0.5114
8	0.9521	0.3711	0.6267	0.3431	0.4761	0.5073
9	0.9035	0.3217	0.6257	0.2686	0.4791	0.5031
10	0.8671	0.3005	0.6345	0.3133	0.4803	0.4945

Table 2. Execution times of algorithms (in seconds) for English data on Nehalem.

m	BAM	BAM2	EBL	LF-BAM2	EA-BAM2	EA-EBL
4	0.6772	0. 9947	0.4573	0.5689	0.3307	0.3431
5	0.6913	0.5511	0.4245	0.5262	0.3203	0.3119
6	0.6567	0.4255	0. 3915	0.4629	0.3214	0.3221
7	0.5942	0.3167	0.3718	0.4065	0.3203	0.3341
8	0.5732	0.2545	0.3511	0.4115	0.3315	0.3472
9	0.5512	0.2025	0.3614	0.3405	0.3411	0.3512
10	0.5441	0.1798	0.4013	0.3792	0.3497	0.3623

Table 3. Execution times of algorithms (in seconds) for Protein data on Nehalem.

The results in Table 3 for protein data show that the EA algorithm scheme is competitive in comparison with the previous algorithms. The EA scheme works best in case of short length patterns of length less than 7. EA-BAM2 is fastest for pattern lengths 4 and 6. EA-EBL performs best for $m = 5$. For the remaining pattern lengths BAM2 performs best. The LF algorithm scheme was not competitive for protein data.

m	BAM	BAM2	EBL	LF-BAM2	EA-BAM2	EA-EBL
4	1.8640	2.3047	1.0800	0.7608	0.8158	0.8468
5	1.8643	1.1992	0.9686	0.7261	0.8082	0.9297
6	1.7281	0.7698	0.9083	0.5581	0.8082	0.9297
7	1.5944	0.6541	0.8801	0.4401	0.7945	0.9921
8	1.3017	0.3906	0.7258	0.3921	0.7192	0.8114
9	1.2395	0.3886	0.7490	0.2929	0.7294	0.8555
10	1.1960	0.3356	0.7636	0.3552	0.7247	0.8994

Table 4. Execution times of algorithms (in seconds) for English data on Haswell.

We also performed the same tests on an Haswell processor (i5-4250U) for English and protein data. The results are shown in Tables 4 and 5. In Table 4 the algorithm LF-BAM2 is a clear winner for all the pattern lengths except for $m = 8$ and 10. For certain pattern lengths such as $m = 4, 5, 6$, the speed up is more than twenty percent. For protein data (Table 5), EBL and BAM2 are the winners. However, LF-BAM2 is better than BAM2 for $m = 4, 5$ and better than EBL for $m = 6, \ldots, 10$.

The Haswell processor has the AVX2 support, which enables 32-byte SIMD computation. We compared the 32-byte version of LF-BAM2 with the 16-byte version for pattern lengths $m = 4, 5, \ldots, 16$. In every case, the 16-byte version was slightly faster.

m	BAM	BAM2	EBL	LF-BAM2	EA-BAM2	EA-EBL
4	0.8944	1.0338	0.4533	0.4776	0.4978	0.4912
5	0.8112	0.6251	0.4096	0.4519	0.4872	0.4880
6	0.7776	0.3541	0.4067	0.3870	0.4929	0.5227
7	0.7430	0.3230	0.4076	0.3467	0.5000	0.5583
8	0.7228	0.2255	0.4078	0.3549	0.5036	0.5676
9	0.7066	0.2271	0.4234	0.3103	0.5096	0.5889
10	0.6952	0.1809	0.4540	0.3478	0.5153	0.6388

Table 5. Execution times of algorithms (in seconds) for Protein data on Haswell.

8 Concluding Remarks

We introduced improved solutions for exact jumbled pattern matching based on the SIMD architecture. These algorithms are an outcome of a long series of experimentation. We developed and tested also some other algorithms using SIMD instructions but only the best are shown in this paper. Especially, it was hard to develop fast jumbled matching algorithms for $m > 16$. It should be realized that if the latency of the used SIMD instructions would improve in future processors, the running times of the algorithms will respectively change.

References

1. G. Benson: Composition alignment. In Proceedings of The 3rd International Workshop on Algorithms in Bioinformatics 2003, pp. 447–461.
2. S. Böcker: Sequencing from compomers: Using mass spectrometry for DNA de novo sequencing of 200+ nt. Journal of Computational Biology 11(6), 2004, pp. 1110–1134.
3. S. Böcker: Simulating multiplexed SNP discovery rates using base-specific cleavage and mass spectrometry. Bioinformatics 23(2), 2007, pp. 5–12.
4. S. Böcker, K. Jahn, J. Mixtacki, J. Stoye: Computation of median gene clusters. Journal of Computational Biology 16(8), 2009, pp. 1085-1099.
5. P. Burcsi, F. Cicalese, G. Fici, Zs. Lipták: On table arrangement, scrabble freaks, and jumbled pattern matching. In Proceedings of the Symposium on Fun with Algorithms 2010, pp. 89–101.
6. P. Burcsi, F. Cicalese, G. Fici, Zs. Lipták: Algorithms for jumbled pattern matching in strings. Int. J. Found. Comput. Sci. 23(2), 2012, pp. 357–374.
7. P. Burcsi, F. Cicalese, G. Fici, Zs. Lipták: On approximate jumbled pattern matching in strings. Theory Comput. Syst. 50(1), 2012, pp. 35–51.
8. D. Cantone, S. Faro: Efficient online Abelian pattern matching in strings by simulating reactive multi-automata. In Proceedings of Prague Stringology Conference 2014, pp. 30–42.
9. T. Chhabra, S.S. Ghuman, J. Tarhio: Tuning algorithms for jumbled matching. In Proceedings of Prague Stringology Conference 2015, pp. 57–66.
10. F. Cicalese, G. Fici, Zs. Lipták: Searching for jumbled patterns in strings. In Proceedings of Prague Stringology Conference 2009, pp. 105–117.
11. E. Domann, T. Hain, R. Ghai, A. Billion, C. Kuenne, K. Zimmermann, T. Chakraborty: Comparative genomic analysis for the presence of potential enterococcal virulence factors in the probiotic enterococcus faecalis strain symbioflor. International Journal of Medical Microbiology 297(7), 2007, pp. 533–539.
12. B. Ďurian, J. Holub, H. Peltola, J. Tarhio: Improving practical exact string matching. Information Processing Letters 110(4), 2010, pp. 148–152.
13. E. Ejaz: Abelian Pattern Matching in Strings. Ph.D. Thesis, Dortmund University of Technology(2010), http://d-nb.info/1007019956.
14. R. Eres, G. M. Landau, L. Parida: Permutation pattern discovery in biosequences. Journal of Computational Biology 11(6), 2004, pp. 1050–1060.

15. S. Faro, T. Leqroc: Smart: String matching algorithms research tool (2015), http://www.dmi.unict.it/∼faro/smart/.
16. S. Grabowski, S. Faro, E: Giaquinta: String matching with inversions and translocations in linear average time (most of the time). Information Processing Letters 111(11), 2011, pp. 516–520.
17. R. Grossi, F. Luccio: Simple and efficient string matching with k mismatches. Information Processing Letters 33(3), 1989, pp. 113–120.
18. M. Hassaballah, S. Omran, Y.B. Mahdy: A review of SIMD multimedia extensions and their usage in scientific and engineering applications. Comput. J. 51(6), 2008, pp. 630–649.
19. A. Hume, D. Sunday: Fast string searching. Software – Practice and Experience 21(11), 1991, pp. 1221–1248.
20. K. Hwang, F.A. Briggs: Computer Architecture and Parallel Processing, McGraw-Hill, 1984.
21. Intel: http://ark.intel.com/products/codename/29896/Lynnfield.
22. 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 (Loaded in Jan. 2016).
23. Intel: Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide.
24. H. Jeong, S. Kim, W. Lee, S.H. Myung: Performance of SSE and AVX instruction sets. CoRR abs/1211.0820 (2012).
25. P. Jokinen, J. Tarhio, E. Ukkonen: A comparison of approximate string matching algorithms. Software – Practice and Experience 26(12), 1996, pp. 1439–1458.
26. S. Karlin: Detecting anomalous gene clusters and pathogenicity islands in diverse bacterial genomes. Trends in Microbiology 9(7), 2001, pp. 335–343.
27. M.O. Külekci: Filter based fast matching of long patterns by using SIMD instructions. In Proceedings of Prague Stringology Conference 2009, pp. 118–128.
28. S. Ladra, O. Pedreira, J. Duato, N.R. Brisaboa: Exploiting SIMD instructions in current processors to improve classical string algorithms. In Proceedings of The 16th East European Conference on Advances in Databases and Information Systems, vol. 7503 of LNCS, Springer, 2012, pp. 254–267.
29. G. Navarro: Multiple approximate string matching by counting. In Proceedings of 4th South American Workshop on String Processing 1997, pp. 95–111.
30. G. Navarro, M. Raffinot: Flexible Pattern Matching in Strings. Practical On-Line Search Algorithms for Texts and Biological Sequences. Cambridge University Press, New York 2002.
31. R. Overbeek, M. Fonstein, M. D'Souza, G.D. Pusch, N. Maltsev: The use of gene clusters to infer functional coupling. In Proceedings of the National Academy of Sciences 96(6), 1999, pp. 2896–2901.
32. H. Peltola, J. Tarhio: Alternative algorithms for bit-parallel string matching. In Proceedings of SPIRE 2003, the 10th International Symposium on String Processing and Information Retrieval, vol. 2857 of LNCS, Springer, 2003, pp. 80–94.
33. A. Salomaa: Counting (scattered) subwords. Bulletin of the European Association for Theoretical Computer Science 81, 2003, pp. 165–179.
34. T. Schmidt, J. Stoye: 2004. Quadratic time algorithms for finding common intervals in two and more sequences. In Proceedings of 15th Annual Symposium of Combinatorial Pattern Matching, vol. 3109 of LNCS, Springer, 2004, pp. 347–358.
35. T. Uno, M. Yagiura: Fast algorithms to enumerate all common intervals of two permutations. Algorithmica 26(2), 2000, pp. 290–309.
36. A. Wiezer, R. Merkl: A comparative categorization of gene flux in diverse microbial species. Genomics 86(4), 2005, pp. 462–475.