On Bit-Parallel Processing of Multi-byte Text

Heikki Hyyrö[1], Jun Takaba[2], Ayumi Shinohara[1,2], and Masayuki Takeda[2,3]

[1] PRESTO, Japan Science and Technology Agency (JST)
`helmu@cs.uta.fi`
[2] Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan
`{j-takaba, ayumi, takeda}@i.kyushu-u.ac.jp`
[3] SORST, Japan Science and Technology Agency (JST)

Abstract. There exist practical bit-parallel algorithms for several types of pair-wise string processing, such as longest common subsequence computation or approximate string matching. The bit-parallel algorithms typically use a size-σ table of match bit-vectors, where the bits in the vector for a character λ identify the positions where the character λ occurs in one of the processed strings, and σ is the alphabet size. The time or space cost of computing the match table is not prohibitive with reasonably small alphabets such as ASCII text. However, for example in the case of general Unicode text the possible numerical code range of the characters is roughly one million. This makes using a simple table impractical. In this paper we evaluate three different schemes for overcoming this problem. First we propose to replace the character code table by a character code automaton. Then we compare this method with two other schemes: using a hash table, and the binary-search based solution proposed by Wu, Manber and Myers [25]. We find that the best choice is to use either the automaton-based method or a hash table.

1 Introduction

Different types of pair-wise string processing algorithms are fundamental in many information retrieval and processing tasks. Let the two processed strings be P and T, of length m and n, respectively. The most basic task is exact string matching, where P is a pattern string and T a text string, and one searches for occurrences of P inside T. Other typical examples include case-insensitive search, regular expression matching, and approximate string comparison. So-called *bit-parallel* algorithms have emerged as practical solutions for several of such string processing tasks. Let w denote the computer word size. We list here some examples of practical bit-parallel algorithms. Each of these has a run time $O(\lceil m/w \rceil n)$. Baeza-Yates and Gonnet [3] proposed an algorithm for exact string matching, and that algorithm can handle also for example case-insensitive search. In [17] Navarro presents methods for allowing repeatable or optional characters in the pattern. Allison and Dix [2], Crochemore et al. [5], and Hyyrö [11] have presented algorithms for computing the length of a *longest common subsequence* between P and T. Myers [15] presented an $O(\lceil m/w \rceil n)$ algorithm for finding approximate occurrences of P from T, when Levenshtein edit distance is used as the measure of similarity. This algorithm can be modified to compute Levenshtein edit distance between P and T [12] as well as to use Damerau edit distance [10].

The above-mentioned, as well as numerous other, bit-parallel algorithms typically use a size-σ table of match bit-vectors, where σ is the size of the alphabet Σ, and the characters in Σ are mapped into the interval $[0 \ldots \sigma - 1]$. Let us call the match table PM. For each character $\lambda \in \Sigma$, the bit values in the corresponding match vector PM_λ mark the positions in P where the character λ occurs. The cost of preprocessing and storing PM is reasonable with small alphabets, such as the 7- or 8-bit ASCII character set. But in case of more general alphabets, perhaps most importantly Unicode text, the range of possible numerical character codes is much larger. To be specific, Unicode character codes fall into the range $[0 \ldots 1114111]$. This makes using a naively stored PM table impractical. A basic observation is that the value PM_λ needs to be explicitly computed only for those $O(m)$ characters λ that occur in P. All other characters share an identical "empty" match vector. One quite straightforward solution is then to store only the non-empty vectors PM_λ into a hash table whose size is $O(m)$ instead of σ. Another solution, similar to the one proposed by Wu, Manber and Myers [25], is to sort into one size-$O(m)$ table the character codes of those λ that occur in P, and store the match vectors PM_λ, in corresponding order, into another size-$O(m)$ table. The value PM_λ is then determined by doing an $O(\log m)$ binary search in the code table (the bound assumes that two character codes can be compared in constant time). If the code of λ is found at the ith position, then the vector PM_λ is in the ith position of the match vector table, and otherwise PM_λ is an empty match vector.

In this paper we propose another approach for storing and locating the PM_λ vectors. The idea is to build an automaton that recognizes the alphabet character codes and whose accepting states identify the corresponding match vectors. The automaton reads the characters in byte-wise manner. We compare this method with the above described alternatives on UTF-8 encoded Unicode text and find that using an automaton is competitive. The results show that the choice of how to handle the match bit-vectors can have a significant effect in terms of the overall processing time: using the binary-search based method of Wu, Manber and Myers [25] may result in the overall processing time being almost three times as much as with the other two methods. We also include a basic direct table lookup in the comparison. This is done by using a restricted multi-byte character set that allows us to use a small table in storing the match vectors. The comparison provides a characterization about the feasibility of using bit-parallel algorithms with Unicode text. This is important as Unicode is becoming more and more widely used. This is true especially on the Internet, which allows people with very different cultures (and character sets) share textual information. XML (eXtensible Markup Language), which is an increasingly popular format for storing data for example on www-pages, uses by default UTF-8 encoded Unicode text. To our best knowledge, the current paper provides the first study about using bit-parallel algorithms in processing multi-byte encoded strings.

2 Unicode

The 7-bit ASCII (American Standard Code for Information Interchange) is a fundamental computer character encoding. It, or some 8-bit extended ASCII form of it, is used with variations of the Latin (or Roman) alphabet. Many common computer systems/programs, such as the UNIX operating systems variants as well as the C programming language, are inherently designed to use such a single-byte ASCII code. For example a zero-byte is typically interpreted to mark an end of file.

In many languages, such as Japanese or Chinese, the commonly used alphabets require a multi-byte character encoding. There are several specialized encodings. For example Japanese Extended Unix Code (EUC), BIG5 (Taiwanese), shift-JIS (Japanese), EUC-KR (Korean), and so on. For compatibility with ASCII-oriented systems, such multi-byte encodings usually reserve the code range $0 \ldots 127$ for the single-byte 7-bit ASCII characters, and the multi-byte characters consist of byte values in the range $128 \ldots 255$. In terms of being able to recognize the characters, an important property in practice is also that the multi-byte code should be a *prefix code*. This means that no character code should be a continuation of another, or conversely, that no character code should be a prefix of another character code.

In order to avoid compatibility problems when processing texts with different languages and alphabets, the Unicode Consortium has created a common international standard character code, Unicode, that can express every character in every language in the world [24, 23]. In its present form, Unicode can express 1114112 different characters. Out of these, currently more than 96000 are actually mapped into some character. Unicode defines a numeric code for each character, but it does not specify how that code is actually encoded. The following three are common alternatives:

UTF-32: A simple fixed-length encoding, where each character is encoded with 4 bytes. The main advantage over UTF-16 and UTF-8 is that processing the text is simple. Downsides are the large space consumption and that different computer architectures may represent multi-byte sequences in different orders ("endianness"), which results in compatibility issues.

UTF-16: A variable-length encoding: each character is encoded by 2 or 4 bytes. The main advantage over UTF-32 and UTF-8 is that the method uses typically only 2 bytes for example for Chinese, Korean or Japanese characters. A downside is that also UTF-16 is affected by the endianness of the used computer architecture.

UTF-8: A variable-length encoding: each character is encoded by 1, 2, 3 or 4 bytes. The main advantage over UTF-32 and UTF-16 is the high level of compatibility: UTF-8 is directly compatible with the ASCII code, and it is not affected by the endianness of the hardware. UTF-8 is the default encoding for the XML format. In terms of space, an advantage is that ASCII characters take only a single byte. A downside in comparison to UTF-16 is that for example Chinese, Korean or Japanese characters take typically 3 bytes.

In this paper we concentrate on UTF-8 as it is the most compatible of these three choices and also serves as an example of a general variable-byte encoding.

In the following we describe the basic structure of UTF-8 encoding. We show the structure of each byte as an 8-bit sequence, where the bit significance grows from right to left, and a value 'x' means that the corresponding bit value is used in storing the actual numeric value of the encoded character. Below each 8-bit sequence we also show the corresponding possible range of numerical (base-10) values for the byte.

```
1 byte:  0xxxxxxx
         0...127
2 bytes: 110xxxxx 10xxxxxx
         192...223 128...191
3 bytes: 1110xxxx 10xxxxxx 10xxxxxx
         224...239 128...191 128...191
4 bytes: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
         240...247 128...191 128...191 128...191
```

The length of a UTF-8 code can be inferred from the most significant (here leftmost) bits of its first byte. If the first bit is zero, the code has a single byte. Otherwise the code has as many bytes as there are consecutive one bits when counting from the most significant bit towards the least significant bit. A byte is a continuation byte of a multi-byte UTF-8 code if and only if its value is in the range 128...191. UTF-8 is clearly a prefix code. The number of available bits ('x') for encoding a character code is 7 for a single-byte code, 11 for a 2-byte code, 16 for a 3-byte code, and 21 for a 4-byte code. Hence UTF-8 encoding can in principle express $2^7 + 2^{11} + 2^{16} + 2^{21} = 2164864$ distinct characters.

3 Basic Variants of String Processing

In this section we review three fundamental and much studied forms of string processing. They were chosen as typical representatives of string processing that can be solved by efficient bit-parallel algorithms. The motivation is to lay basic background: The discussed tasks are the ones we will concentrate on in the tests with multi-byte encoded text in Section 5. But let us first introduce some further basic notation. The length of a string A is denoted by $|A|$, A_i is the ith character of A, and $A_{i..j}$ denotes the *substring* of A that begins from its ith character and ends at the jth character. If $j < i$, we interpret $A_{i..j}$ to be the empty string ϵ. If A is nonempty, the first character of A is A_1 and $A = A_{1..|A|}$. The substring $A_{1..j}$ is a *prefix* and the substring $A_{j..|A|}$ is a *suffix* of A. The string C is a *subsequence* of the string A if C can be derived by deleting zero or more characters from A. Thus C is a subsequence of A if the characters $C_1 \ldots C_{|C|}$ appear in the same order, but not necessarily consecutively, in A.

Exact String Matching. Exact string matching is one of the most fundamental string processing tasks. When one is given a length-m pattern string P

and a length-n text string T, the task is to find all text indices j for which $P = P_{1..m} = T_{j-1+m..j}$. A common variant of this, and also the following two other tasks, is *case insensitive matching*, where no distinction is made between lower- and uppercase characters.

Longest Common Subsequence. The string C is a *longest common subsequence* of the strings P and T, if C is a subsequence of both P and T, and no longer string with this property exists. We denote a longest common subsequence between the strings P and T by LCS(P, T), and LLCS(P, T) denotes the length of LCS(P, T). Both LCS(P, T) and LLCS(P, T) convey information about the similarity between P and T. This may be used for example in molecular biology (see e.g. [21]), file comparison (e.g. the Unix "diff" utility), or assessing how closely related two words are to each other (e.g. [20]).

Edit Distance and Approximate String Matching. Edit distance is another measure of similarity between two strings. The edit distance $ed(P, T)$ between the strings P and T is in general defined as the minimum number of edit operations that are needed in transforming P into T or vice versa.

The task of *approximate string matching* is to find all text locations where a text substring is within a given edit distance from the pattern. A more formal definition is that the task is to find all text indices j for which $ed(P, T_{h..j}) \leq k$, where $h \leq j$ and k is the given error threshold.

Above we did not specify the type of the edit distance. The following distances are typical. We denote by $ed_s(P, T)$ a simple edit distance that allows one edit operation to delete or insert a single character. The values $ed_s(P, T)$ and LLCS(P, T) are connected by the equality $2\times$LLCS$(A, B)= n + m - ed_{id}(A, B)$ (e.g. [6]). Probably the most common form of edit distance is Levenshtein edit distance [14], which extends the simple edit distance by allowing also the operation of substituting a single character with another. We denote Levenshtein edit distance between P and T by $ed_L(P, T)$. Damerau distance [8], which we denote by $ed_D(P, T)$, is used especially in spelling correction related applications. It extends Levenshtein distance by allowing also a fourth edit operation: transposing (swapping) two adjacent characters.

3.1 Bit-Parallel Algorithms

During the last two decades, so-called *bit-parallel algorithms* have emerged as practical choices for several string processing tasks. The principle of such algorithms is in general to take advantage of the fact that computers process data in chunks of w bits, where w is the computer word size (in effect the number of bits in a single register within the processor). Currently most computers have a word size $w = 32$, but also the word size $w = 64$ is becoming increasingly common. In addition, most current personal computers support specialized instruction extension sets, such as MMX or SSE, that allow one to use $w = 64$ or even $w = 128$. Bit-parallel algorithms store several data-items into a single computer word, and then update them in parallel during a single computer operation.

```
**ComputePM**($P$)
  **For** $\lambda \in \Sigma$ **Do** $PM_\lambda \leftarrow 0^m$
    **For** $i = 1 \ldots m$ **Do** $PM_{P_i} \leftarrow PM_{P_i} \mid 0^{m-i}10^{i-1}$

**Bit-ParallelProcessing**($P, T$)
  **ComputePM**($P$)
  **InitializeVectors**
  **For** $j = 1 \ldots n$ **Do**
    **ProcessVectors**($T_j$)
  **ProcessResult** (if required)
```

Fig. 1. Preprocessing the PM-table and a basic skeleton for the discussed bit-parallel algorithms

We use the following notation with bit-vectors: '&' denotes bitwise "and", '|' denotes bitwise "or", '^' denotes bitwise "xor", '~' denotes bit complementation, and '<<' and '>>' denote shifting the bit-vector left and right, respectively, using zero filling in both directions. Bit positions are assumed to grow from right to left, and we use superscripts to denote repetition. As an example let $V = 1110010$ be a bit vector. Then $V[1] = V[3] = V[4] = 0$, $V[2] = V[5] = V[6] = V[7] = 1$, and we could also write $V = 1^3 0^2 10$.

The general high-level scheme for bit-parallel string processing algorithms is as follows. First a size-σ match table PM is computed for the length-m string P. PM holds a length-m match bit-vector PM_λ for each character $\lambda \in \Sigma$. The bit-vector PM_λ identifies the positions in the string P where the character λ occurs: the ith bit of PM_λ is set if and only if $P_i = \lambda$. For simplicity, we will assume throughout this paper that $m \le w$. The case $m > w$ can be handled by simulating a length-m bit-vector by concatenating $\lceil m/w \rceil$ length-w bit-vectors, and thus the table PM occupies in general $\sigma \lceil m/w \rceil$ bits. Once PM is preprocessed and the data bit-vectors used by the algorithm have been initialized, the bit-parallel algorithm processes the string T sequentially. At each character T_j the algorithm updates the data bit-vectors by using bit-operations. Depending on the task, the algorithm may at this point also update some score value and/or check whether a match was found at position j. Fig. 1 shows pseudocode for preprocessing PM and a skeleton for the actual processing phase. The sub-procedure "ProcessVectors" encloses all steps that a particular algorithm conducts at character T_j. In what follows we will show some specific choices for the sub-procedures. Each bit-parallel algorithm that we discuss runs in $O(n)$ time when $m \le w$ and in general in $O(\lceil m/w \rceil n)$ time. But a detailed discussion of any of these algorithms is outside the scope of this paper; the reader should look into the given references for more information about them. The algorithms are shown as examples of different types of bit-parallel algorithms, and they are the ones we use in testing.

Due to the nature of the match table PM, it is a well-known fact that bit-parallel algorithms can be easily modified to be case-insensitive. This is usually said more broadly: the algorithms can use *classes of characters*. For each character λ we may define a set of characters that are deemed to match with λ. This can be done simply by setting the ith bit of $PM_{\lambda'}$ for all such λ' for which we wish to define $P_i = \lambda = \lambda'$.

```
InitializeVectors-SA
    $R \leftarrow 0^m$

ProcessVectors-SA($T_j$)
    $R \leftarrow ((R << 1) \mid 0^{m-1}1)$ & $PM_{T_j}$
    If $R$ & $10^{m-1} \neq 0^m$ Then
        Report an occurrence of
        $P$ ending at $T_j$.
```

```
InitializeVectors-LLCS
    $P \leftarrow 1^m$

ProcessVectors-LLCS($T_j$)
    $X \leftarrow P$ & $PM_{T_j}$
    $P \leftarrow (P + X) \mid (P - X)$

ProcessResult-LLCS
    LLCS($P, T$) is the number of
    zero bits in $P$
```

```
InitializeVectors-ASM
    $currDist \leftarrow m$
    $VN \leftarrow 0^m$
    $VP \leftarrow 1^m$

ProcessVectors-ASM($T_j$)
    $D0 \leftarrow (((PM_{T_j}$ & $VP) + VP)$ ^ $VP) \mid PM_{T_j} \mid VN$
    $HP \leftarrow VN \mid \sim (D0 \mid VP)$
    $HN \leftarrow D0$ & $VP$
    If $HP$ & $10^{m-1} = 10^{m-1}$ Then
        $currDist \leftarrow currDist + 1$
    Else If $HN$ & $10^{m-1} = 10^{m-1}$ Then
        $currDist \leftarrow currDist - 1$
    If $currDist \leq k$ Then
        Report an approximate occurrence of $P$ at $T_j$
    $VP \leftarrow (HN << 1) \mid \sim (D0 \mid (HP << 1))$
    $VN \leftarrow D0$ & $(HP << 1)$
```

Fig. 2. Bit-parallel procedures for exact string matching (*upper left*), LLCS computation (*lower left*), and approximate string matching (*right*)

Baeza-Yates and Gonnet proposed the bit-parallel shift-and algorithm [3] for exact string matching. When $m \leq w$, its behaviour is similar, although much faster, than that of the well-known linear-time string matching algorithm of Knuth, Morris and Pratt [13]. Shift-and processes all text characters in sequential order, and thus it is typically somewhat slower than algorithms that try to skip quickly over such text areas that are seen not to contain a match (e.g. [4, 7, 18]). The latter approach is, however, more difficult in the case of variable-length encoded text. The pseudocode for the bit-parallel processing of the shift-and algorithm at the character T_j is shown in the upper left part of Fig. 2.

Allison and Dix [2] proposed the first bit-parallel algorithm for the longest common subsequence problem. To our best knowledge, this was also the first bit-parallel approximate string processing algorithm. Later Crochemore et al. [5] and Hyyrö [11] have proposed similar variants. The lower left part of Fig. 2 shows the pseudocode for the bit-parallel LLCS processing of [11] that makes four operations per character of T. As discussed in [11], these bit-parallel algorithms are very practical for LLCS-computation.

Myers [15] presented an efficient bit-parallel algorithm for approximate string matching under Levenshtein edit distance. The tests in [16] show that this algorithm is in many cases the fastest in practice. Here we refer to so-called "verification capable" algorithms that are based on actually computing edit distance. It is easy to transform Myers' algorithm to compute edit distance [12], and it has also been modified to use Damerau distance [10]. The right side of Fig. 2 shows the pseudocode for the slightly simpler variant of Hyyrö [9, 19].

4 Storing the Match Vectors

As discussed in Section 1, storing the match vectors PM_λ into a size-σ table is not practical in the case of Unicode encoded text or similar large alphabets. In this section we first propose an approach that uses a code automaton to overcome this problem. Then we also discuss two other options.

Code Automaton. Our proposal is to build a minimized code automaton that uniquely recognizes the encoding of each character that appears in P, and in addition accepts the encodings of all those characters that do not appear in the pattern. Let u be the number of different characters that appear in P. Then the code automaton has $u + 1$ accepting states: one for each different character in P, and one that represents all other characters in Σ. If the character λ appears in P, we associate PM_λ with the state that accepts the encoding of λ. The state that represents those characters that do not appear in P will be associated with a zero match vector 0^m. In our case of multi-byte character encoding, we will read the text T with the code automaton one byte at a time. Whenever the automaton recognizes a character, a bit-parallel algorithm can process the currently read text character T_j by using the match vector that is associated with the current accepting state.

Such a code automaton can conceptually be built by first composing a trie over the encodings of all characters in the alphabet, and then minimizing it so that all leaves that correspond to characters that do not appear in the pattern P are merged into a single leaf. The leaves corresponding to the characters that appear in the pattern are not merged. When u has the same meaning as above, the resulting DAG (Directed Acyclic Graph) has $u + 1$ leaf nodes, which are the accepting states of the corresponding automaton. The final automaton is then composed by augmenting the DAG with Aho-Corasick *failure links* [1] and associating the match vectors with the accepting states. The process (except for the match vectors) is similar to how the pattern matching automaton used in [22] is built. The main difference is that here the "set of patterns" of the pattern matching automaton is formed by those character encodings that appear in P. Fig. 3 shows an example.

Hash Table. The second approach is to use a hash table, which is a standard text-book procedure for storing keys. In this scheme the range of numerical values of the character encodings (for example $1 \ldots 1114112$ in the case of the full range of Unicode encodings) is mapped onto a relatively small integer range $1 \ldots x$. Let $code(\lambda)$ denote the numerical value of the encoding of the character λ, and let the function $hash(code(\lambda))$ give the mapping onto the range $1 \ldots x$. For each λ that occurs in P, the value $code(\lambda)$ is stored into the position $hash(code(\lambda))$ of the match vector table. If two non-equal characters in P have the same mapping, different mechanisms can be used. We describe here a simple linear hashing scheme. If the position $hash(code(\lambda))$ in the table is already used when we are attempting to store $code(\lambda)$ into it, we continue probing the next positions one-by-one until an empty position is found and store the value there. If the end of the table is reached, we continue from the first position of the table. This works as long as the table is not yet full, but the process takes h steps in the worst case, where h is the number of items currently in the table. But the scheme works well if the number of stored items is small in comparison to x. The match vectors are

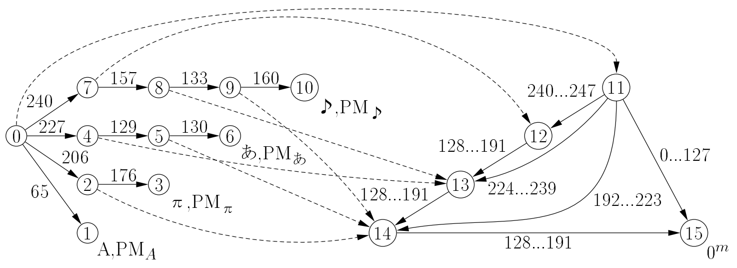

Fig. 3. An example of a code automaton for UTF-8. Here the pattern has characters with bytewise decimal UTF-8 encodings 65 (= ‘A’), 206 176 (= ‘π’), 227 129 130 (’o’ in Japanese hiragana), and 240 157 133 160 (= a note symbol). The corresponding accepting states are numbered 1, 3, 6 and 10, respectively. State 0 is the initial state, and state 15 is the accepting state for those characters that do not appear in the pattern. The matched pattern character and the corresponding match vector are shown next to each accepting state. The dashed arrows correspond to failure links that are followed when the current state does not have an outgoing solid arrow for the current byte value. After reaching an accepting state, the automaton resets itself into state 0 (this is an empty transition)

associated with the corresponding character encodings in the table. Finding the encoding value of a text character T_j from the table works in similar fashion: first the mapping value $hash(code(T_j))$ is computed, and then the table is checked from the corresponding position onwards until either the value $code(T_j)$ or an empty position is found. In the former case we use the associated match vector. In the latter case the table does not contain $code(T_j)$, and we use an empty match vector 0^m.

In our case we know in advance that the table will hold exactly u values, where u is again the number of distinct characters in P. For efficiency we use an extended table of size $x + u$ so that we do not need to worry about reaching the end of the table. With multi-byte text we have tested a very simple mapping. It maps a multi-byte code onto the range $0 \ldots 255$ (corresponds to $x = 256$) by using the value of the last byte in the code. As far as the encodings are random enough not to share too many identical last bytes, this works very efficiently.

Binary Search. The third approach is derived from the proposition of Wu, Manber and Myers [25]. In it the numerical values of the character encodings of the u distinct pattern characters are stored into a size-u table, and the values are sorted. The match vectors are associated with the corresponding values. The value $code(T_j)$ is looked up from the table by doing an $O(log_2 u)$ binary search. Again we use the corresponding match vector if the value $code(T_j)$ was found from the table, and otherwise an empty match vector 0^m.

AMD Athlon64												
	Shift-and				LCS				ASM			
m	4	8	16	32	4	8	16	32	4	8	16	32
AUT	128	127	127	128	129	128	129	130	114	113	112	113
BIN	209	248	293	380	199	232	270	351	167	192	219	278
HASH	120	122	130	150	122	126	133	154	109	110	114	129
SIMP	100	100	100	100	100	100	100	100	100	100	100	100

Intel Pentium 4											
Shift-and				LCS				ASM			
4	8	16	32	4	8	16	32	4	8	16	32
84	84	86	87	99	89	90	90	97	95	95	95
185	221	269	323	205	223	269	322	192	218	252	302
109	112	121	138	104	96	104	122	101	100	105	121
100	100	100	100	100	100	100	100	100	100	100	100

AlphaStation XP1000												
	Shift-and				LCS				ASM			
m	4	8	16	32	4	8	16	32	4	8	16	32
AUT	140	141	140	141	131	132	132	133	131	132	132	132
BIN	220	256	299	354	196	224	251	293	182	211	246	285
HASH	114	118	125	145	113	115	118	138	111	114	120	135
SIMP	100	100	100	100	100	100	100	100	100	100	100	100

Sparc Ultra 2											
Shift-and				LCS				ASM			
4	8	16	32	4	8	16	32	4	8	16	32
103	103	106	107	108	110	113	115	103	104	105	106
218	245	277	310	202	229	258	289	168	186	206	230
103	104	108	119	106	107	109	119	100	102	104	111
100	100	100	100	100	100	100	100	100	100	100	100

Fig. 4. The results for the three tested string processing tasks on four different computer architectures. *AUT*: code automaton, *BIN*: binary search, *HASH*: hash table, *SIMP*: direct table lookup (the multibyte characters allowed in the patterns were restricted to a small subset, thus allowing to use a simple table)

5 Test Results

We implemented and tested the three match table handling schemes from the previous section. In order to characterize their performance in conjunction with bit-parallel algorithms of various complexity, we did separate tests with each of the three bit-parallel algorithms discussed in Section 3.1. In order to evaluate hardware-dependency, we tested on four different computers: AMD Athlon64, Intel Pentium 4, AlphaStation XP1000 and Sparc Ultra 2. The code was exactly the same with all computers, and the different bit-parallel methods used the same file-handling framework. On the AlphaStation we used CC compiler, and on the other computers we used GCC. All code was compiled with the "-O9" optimization switch. The tested strings were UTF-8 encoded, and they were generated randomly. The lengths of P were $m = 4$, 8, 16 and 32. The length of T was at least one million characters in the case of searching, and P and T were of equal length in the case of computing LLCS(P, T). Each test included 100 different choices for P. In searching we used a single text T, and in computing the value LLCS(P, T) we used as many T as was necessary for their combined length to be at least one million characters. In order to estimate the overhead of these match vector handling methods in comparison to the simple lookup from a size-σ table, we included also a test where the strings contain only UTF-8 characters that have distinct last byte values. This way our simple hash table method could be turned into a direct table lookup. Fig. 4 shows the results as a percentage of the running time of the direct table lookup.

In each case, using binary search was clearly the worst method on all computers. In some cases the overall processing time was almost three times longer than with the code automaton. The relative performance of the hash table and the code automaton varied depending on m and the computer. On Pentium 4 the code automaton was always the fastest scheme, in fact even faster than the direct table lookup. We re-checked this with another compiler, and the situation remained the same. This is perhaps due to some pipelining effect etc. We note that this does not depend on the fact that the direct table lookup used restricted character encodings: we tested also the other schemes on the specially encoded strings, and their running times were practically the same as with the regular random strings. On Sparc and AMD the automaton and hash table performed fairly equally. With small m the hash table tended to be often a little faster (always less than 10%), and with larger m the code automaton became the better of the two. On AlphaStation the hash table was up to roughly 20% faster than the code automaton, but still a little bit slower with $m = 32$.

One conclusion is that the code automaton is typically very competitive against the other methods. We also note that the overall penalty for not being able to use a direct table lookup is reasonably small: never more than roughly 40%. Since the advantage of the bit-parallel methods over other kinds of algorithms is often much larger than this, they seem to be practical also with multi-byte encoded text. In addition, also the other types of string processing algorithms will have to pay some penalty for having to deal with multi-byte encoding. We also point out that the automaton is quite insensitive to the value of m or the properties of the strings. Hence it is a feasible option for use with bit-parallel multi-byte string processing.

6 Conclusions

In this paper we proposed a scheme that uses a code automaton for looking up match vectors of multi-byte encoded characters. We also discussed two other schemes for the same task, and compared the three quite extensively. The test results showed that using the automaton is often the fastest choice, and never more than roughly 25% slower than the next best of these schemes. The binary search based method proposed by Wu, Manber and Myers in [25] was found to perform very slow. Using it resulted always in the longest processing time, in one case almost three times longer than when using the code automaton. Overall the test results give an idea about the feasibility of processing multi-byte encoded text with bit-parallel algorithms. As the test indicated the penalty to be at most roughly 40%, bit-parallel algorithms are a viable option with multi-byte text.

References

1. Aho, A., Corasick, M.: Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.
2. Allison, A., Dix, T.L.: A bit-string longest common subsequence algorithm. *Information Processing Letters*, 23:305–310, 1986.
3. Baeza-Yates, R., Gonnet, G.: A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.
4. Boyer, R. S., Moore, J. S.: A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.
5. Crochemore, M., Iliopoulos, C. S., Pinzon, Y. J., Reid, J.F.: A fast and practical bit-vector algorithm for the longest common subsequence problem. *Information Processing Letters*, 80:279–285, 2001.

6. Crochemore, M., Rytter, W.: *Text Algorithms.* Oxford University Press, Oxford, UK, 1994.
7. Czumaj, A., Crochemore, M., Gasieniec, L., Jarominek, S., Lecroq, T., Plandowski, W., Rytter, W.: Speeding up two string-matching algorithms. *Algorithmica*, 12:247–267, 1994.
8. Damerau, F.: A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.
9. Hyyrö, H.: Explaining and extending the bit-parallel approximate string matching algorithm of Myers. Technical Report A-2001-10, Dept. of Computer and Information Sciences, University of Tampere, Tampere, Finland, 2001.
10. Hyyrö, H.: Bit-parallel approximate string matching with transposition. In *Proc. 10th International Symposium on String Processing and Information Retrieval (SPIRE'2003)*, LNCS 2857, 2003.
11. Hyyrö, H.: Bit-parallel LCS-length computation revisited. In *Proc. 15th Australasian Workshop on Combinatorial Algorithms (AWOCA 2004)*, 2004.
12. Hyyrö, H., Navarro, G.: Faster bit-parallel approximate string matching. In *Proc. 13th Combinatorial Pattern Matching (CPM'2002)*, LNCS 2373, 2002.
13. Knuth, D. E., Morris, J. H. Jr, Pratt, V. R.: Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.
14. Levenshtein, V.: Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.
15. Myers, G.: A fast bit-vector algorithm for approximate string matching based on dynamic progamming. *Journal of the ACM*, 46(3):395–415, 1999.
16. Navarro, G.: A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
17. Navarro, G.: NR-grep: a fast and flexible pattern matching tool. *Software Practice and Experience*, 31:1265–1312, 2001.
18. Navarro, G., Raffinot, M.: Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithms*, 5(4), 2000.
19. Navarro, G., Raffinot, M.: *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, 2002.
20. Robertson, A. M., Willett, P.: A comparison of spelling-correction methods for the identification of word forms in historical text databases. *Literary and Linguistic Computing*, 8(3):143–152, 1993.
21. Sankoff, D., Kruskal, J. (eds.): *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley, 1983.
22. Takeda, M., Miyamoto, S., Kida, T., Shinohara, A., Fukumachi, S., Shinohara, T., Arikawa, S.: Processing text files as is: Pattern matching over compressed tests, multi-byte character texts, and semi-structured tests. In *Proc. 9th International Symposium on String Processing and Information Retrieval (SPIRE'2002)*, LNCS 2476, 2002.
23. Unicode Consortium.: Unicode Home Page, http://www.unicode.org/.
24. Unicode Consortium.: *The Unicode Standard 4.0.* Addison-Wesley, 2003.
25. Wu, S., Manber, U., Myers, E.: A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.