

# TARA: An Algorithm for Fast Searching of Multiple Patterns on Text Files

M. Oğuzhan Külekci

Turkish Army Gendarme Headquarter

Beştepe, Ankara, TURKEY 41470

Email: kulekci@su.sabanciuniv.edu.tr

**Abstract**—This work introduces a new multi-pattern matching algorithm that performs searching of fixed-length strings on text files very fast by benefiting from bit-parallelism. The algorithm is given name *tara*. Bounded gaps as well as character classes in keywords are also supported. Although the worst case time complexity is quadratic, it performs very fast in practise. Experiments are conducted to compare the performance of the proposed algorithm with widely used *GNU grep* file search utility and also with 9 variants of Aho&Corasick and Comentz&Walter algorithms on natural language text. On the average *tara* is approximately 10% faster than *grep*, where up to 70% percent speed up is observed. The benchmark with the AC and CW variants results that the speed up obtained by *tara* is 3,5 times relative to its nearest successor.

## I. INTRODUCTION

Searching for keywords on text files is a very common task that we all experience in daily life. Numerous algorithms for locating exact occurrences of both single and multiple patterns in a given text have been proposed during the last three decades. Many of the well-known *single* pattern matching algorithms are described in [1]. The performance analysis along with a taxonomy of *multi-pattern* matching algorithms can be found in [2], [3], and [4].

Studies aiming efficient handling of the multiple pattern case have mainly focused on automata theoretic approaches. The basic linear-time algorithm, which was introduced by Aho&Corasick (AC) [5], creates a finite state machine from the given patterns at the beginning, and then scans each character of the text once via this automaton. Following that work, Commentz-Walter [6] combined the idea of Boyer-Moore's [7] single pattern matching algorithm with the AC machine and proposed a sublinear solution to the problem. Similarly, Baeza-Yates [8] integrated the Horspool's [9] method, and Crochemore *et al.* [10] applied the reverse factor algorithm to the AC machine.

Besides the automata theoretic approaches, some researchers considered to benefit from bit parallelism by representing patterns, text or states of the finite state machine in bit strings. Although bit-parallel approaches (e.g. shift-or algorithm [11]) are very powerful in single pattern search, it is more difficult to apply bit-parallelism in multi-pattern case. Baeza-Yates&Gonnet [11] proposed searching via simple arithmetic and logical operations by representing states of search with numbers provided that the computer word size is large enough to include all keywords in the pattern set.

Kim&Kim [12] designed an algorithm that encodes the patterns and text initially, and performs scanning with bitwise hashing operations. Another significant work on the topic is the Wu&Manber's [13] multi-pattern matching algorithm that again exploits the Boyer-Moore's idea with an efficient hashing scheme.

*Tara* is yet a new bit-parallel algorithm for multiple fixed-length pattern searching. The algorithm scans the text in factors equal to the maximum length of input patterns. The idea is to check occurrences of all patterns together in the investigated window with minimum number of bitwise operations and perform the largest possible shifts while passing over the text.

In practice, while searching for strings, the ability of searching patterns with don't care symbols and character classes are also important. Proposed algorithm handles these cases as well. As an example, the Turkish word *çığlık* (*scream*) may be observed as *cıglik* in a file that does not support Turkish characters *ç*, *ı*, and *ğ*. Thus, sometimes it may be useful to perform the search by using character classes in the pattern, such as  $[çc][ıı][ğg]l[ıı]k$ , which permits alternative characters at some positions. Similarly, one may need to use wild characters at some positions that he/she does not care about.

The performance of *tara* is compared with the well-known and most widely used search utility program *grep*<sup>1</sup> that is being developed since 1993 under the GNU project, and also with AC and CW variants. The experiments on 8500 test pattern sets showed that especially for small number of patterns *tara* is approximately 1.5 times faster than *grep*, and 5 times than the nearest variant of AC and CW.

## II. THE TARA ALGORITHM

Let  $P = \{p_0, p_1, \dots, p_{m-1}\}$  be the set of  $m$  patterns that are to be scanned in text  $T[0 \dots n - 1]$  of  $n$  characters, and  $LP = \{lp_0, lp_1, \dots, lp_{m-1}\}$  be the corresponding lengths of patterns in  $P$ . The alphabet is denoted by  $\Sigma$ . Maximum and minimum values of  $LP$  are stored in *maxlen* and *minlen* variables. The algorithm is explained below on an example where it is assumed that  $P = \{bal, peynir, re[çç]el\}$ ,  $LP = \{3, 6, 5\}$ , *maxlen* = 6, and *minlen* = 3. Note the inclusion

<sup>1</sup>The version 2.5.1a of *grep* that can be downloaded from <http://gnuwin32.sourceforge.net/packages/grep.htm> or [www.gnu.org/software/grep](http://www.gnu.org/software/grep) is used within this study.

POSITION						Item ID
0	1	2	3	4	5	
b	a	l				...
	b	a	l			...
		b	a	l		...
			b	a	l	...
p	e	y	n	i	r	...
r	e	{c, ç}	e	l		...
	r	e	{c, ç}	e	l	...

TABLE I

THE ALIGNMENT MATRIX GENERATED FOR THE SAMPLE PATTERN SET  $P$ .

of character class  $[cç]$  in the third pattern of set  $P$ , which demonstrates this character may be  $c$  as well as  $ç$ .

### A. Overview of the algorithm

The main idea is; while sliding a window of size  $maxlen$  on the text, checking for any occurrences of given patterns in the window very fast via bitwise techniques, and performing maximum shift according to the visited characters and the first character<sup>2</sup> succeeding the current window. The preprocessing of the algorithm consist of computing *Alignment*, *Mask*, *Shift* matrices and the *scan order* of the positions in the window.

Given the set of patterns, *tara* first finds the maximum ( $maxlen$ ) and minimum ( $minlen$ ) length of the input strings and creates an alignment matrix that includes all possible placements of input patterns totally fitting in a window of size  $maxlen$ . Each row of the alignment matrix represents a unique possible placement of a given pattern in the window. By using that alignment matrix, the mask and shift matrices that are to be used in the main search loop are computed. Actually,  $ShiftMatrix[ch][pos]$  indicate the amount of shift for next attempt if character  $ch$  is observed at position  $pos$  and  $MaskMatrix[ch][pos]$  is a bit vector where the bits indicate which of the rows of the alignment matrix are appropriate candidates of match if one encounters  $ch$  at position  $pos$ . The window under investigation is traversed according to a scan order that is computed beforehand.

### B. Alignment Matrix

Alignment matrix holds all possible placements of each pattern wholly residing in a window of size  $maxlen$ . Pattern  $p_i$  may begin at position  $j$ , if  $(j + lp_i) \leq maxlen$ , where  $0 \leq j < maxlen$ . Hence, there are  $(maxlen - lp_i + 1)$  such possible placements for each pattern  $p_i$ , and total number of items in the alignment matrix, which will be referred as *itemcount* from now on, is:

$$itemcount = \sum_{i=0}^{m-1} maxlen - lp_i + 1$$

The alignment matrix created for the sample patterns is given in Table I. The words  $p_0$  (bal),  $p_1$  (peynir), and  $p_2$  (re[cç]el) start at positions  $\{0, 1, 2, 3\}$ ,  $\{0\}$ , and  $\{0, 1\}$

<sup>2</sup>That is with the same idea of Sunday's quick search [14] algorithm, where the first succeeding character after the current window is also considered while computing the shift amount as it will exist in next attempt for sure.

CH	POSITION					
	0	1	2	3	4	5
b	1001111	0001110	0001100	0001001	0000011	0100111
a	1001110	0001101	0001010	0000101	0001011	0100111
l	1001110	0001100	0001001	0000011	0100111	1101111
p	1011110	0001100	0001000	0000001	0000011	0100111
e	1001110	0111100	1001000	0100001	1000011	0100111
y	1001110	0001100	0011000	0000001	0000011	0100111
n	1001110	0001100	0001000	0010001	0000011	0100111
i	1001110	0001100	0001000	0000001	0010011	0100111
r	1101110	1001100	0001000	0000001	0000011	0110111
c	1001110	0001100	0101000	1000001	0000011	0100111
ç	1001110	0001100	0101000	1000001	0000011	0100111
...	1001110	0001100	0001000	0000001	0000011	0100111

TABLE II

THE MASK MATRIX GENERATED FOR THE SAMPLE PATTERN SET  $P$ .

Bit Index	6	5	4	3	2	1	0
Value	1	0	0	0	0	1	1

TABLE III

$MaskMatrix[e][4]$

respectively. Each row of the alignment matrix is named as an *item*, and identified by its row index starting from 0. There are seven items in the sample alignment matrix for the given three patterns.

### C. Mask Matrix

Mask matrix consist of  $|\Sigma|$  rows and  $maxlen$  columns. Each cell contains a mask value composed of *itemcount* number of bits. The rows represent the characters of the alphabet, and columns represent the positions. The bit stream in  $MaskMatrix[ch][pos]$  indicates which of the items from the *AlignmentMatrix* are candidates of match, if one encounters that  $ch$  at that  $pos$ . The  $i^{th}$  bit is 1, if  $i^{th}$  item of the alignment matrix is appropriate at the given conditions, and it is 0 otherwise.

The mask matrix generated for the sample pattern set  $P$  is given in Table II. Let's investigate the bit stream at  $MaskMatrix[e][4]$ , which is equal to 1000011, assuming that the window is placed at text position  $T[j \dots j + 5]$ . The bit indices with the corresponding values are sketched in Table III. The bits at positions 0, 1, and 6 are equal to 1, which means items from the *AlignmentMatrix* at row 0, 1, and 6 are appropriate if one detects character  $e$  at  $T[j + 4]$ . The alignment of those possible candidates are shown in Table IV. Note that the placements of pattern  $bal$  at  $T[j \dots j+2]$  and  $T[j+1 \dots j+3]$  are also valid candidates, as character at  $T[j+4]$  does not give any evidence about their existence or nonexistence. However, the remaining 4 items from the alignment matrix are exactly out of interest for the current window, as they require characters different from  $e$  at  $T[j + 4]$ .

...	T[j]	T[j+1]	T[j+2]	T[j+3]	T[j+4]	T[j+5]	...
...					e		...
	b	a	l				
		b	a	l			
		r	e	[cç]	e	l	

TABLE IV

POSSIBLE PLACEMENTS OF THE SAMPLE PATTERNS ACCORDING TO  $MaskMatrix[e][4]$ .

CHARACTER	POSITION							
	0	1	2	3	4	5	6	7
b	1	2	3	4	1	2	3	
a	1	2	3	4	5	1	2	
l	1	2	3	4	5	6	1	
p	1	1	2	3	4	5	6	
e	1	2	1	1	2	1	2	
y	1	2	3	1	2	3	4	
n	1	2	3	4	1	2	3	
i	1	2	3	4	5	1	2	
r	1	2	1	2	3	4	1	
c	1	2	3	4	1	2	3	
ç	1	2	3	4	1	2	3	
OTHERS	1	2	3	4	5	6	7	

TABLE V

THE SHIFT MATRIX GENERATED FOR THE SAMPLE PATTERN SET  $P$ .

#### D. Shift Matrix

Shift matrix consist of  $|\Sigma|$  rows and  $maxlen + 1$  columns. The amount of minimum safe shift values, when a character  $ch$  is observed at position  $pos$ , where  $ch \in \Sigma$  and  $0 \leq pos \leq maxlen$ , are stored in  $ShiftMatrix$ . Note that the first character following the current window is also taken into consideration as in quick search algorithm of Sunday [14]. The shift matrix calculated for the sample patterns is given in Table V.

The main idea of the shift matrix used in the algorithm is a generalization of the Sunday's quick search bad character shift function to multi-pattern case. Let's say character  $r$  is observed at position  $j + 4$  in an attempt of searching text portion  $T[j..j+5]$  for the sample patterns. The window should be shifted to right at least by 3 characters so that  $r$  coincides with the first character of the 6th item ( $re[cç]e1$ ) from the alignment matrix. Thus, the occurrence of  $r$  at  $T[j+4]$  indicate that  $T[j+3..j+8]$  should be investigated in the next attempt. While checking the text portion  $T[j..j+5]$  every visited character returns a such shift value and the window is slid right by largest of those values for the next attempt.

A character at a position  $pos$  should have a shift value of  $(1 + pos)$ , if it is not observed in any of the input patterns. Thus, every cell in the shift matrix is set to one plus its column index at the beginning. If the occurrence of a character  $ch$  at a position  $pos$  indicates an evidence for the alignment of a pattern other than the ones included in the current window, then the shift value is computed accordingly. For example, if one observes character  $b$  at position 4, it is possible that the word  $bal$  resides between positions 4 to 6.

Thus,  $ShiftMatrix[b][4]$  should be 1, as sliding the window by one character fits with this alignment of the word. Note that a character may coexist in more than one pattern, and result in different shift values. The minimum of those possible values is selected to assign safe shifts correctly.

#### E. Scan Order

Scan order defines the order of characters to be visited while searching for the occurrences of patterns in the window of size  $maxlen$ . Let's assume text  $T[j..j + maxlen - 1]$  is the window under investigation. The characters at positions  $s = j - 1 + k \cdot minlen$  for  $k = 1, 2, 3 \dots$ , where  $s \leq j + maxlen - 1$ , are *mandatory* check points that must be visited to make sure whether any of the patterns exist in the specified region. Thus, the characters at these mandatory indices should be checked first to decide fast in case of a mismatch. The positions other than these mandatory points are traversed with the aim of achieving maximum shift values. As the shift values tend to be larger at the right hand side, the remaining positions are checked from right-to-left. Hence, the scan order computed for the sample patterns is  $ScanOrder = \{2, 5, 4, 3, 1, 0\}$ .

#### F. Main Search Loop

The pseudo code of the whole algorithm is given in Figure 1. Note that the input text is padded with  $maxlen$  number of null characters to be able to locate the search window at the end of the text. The investigation of a  $maxlen$  size window begins by initializing the  $flag$  and  $jump$  variables. Every item in the alignment matrix is a possible candidate at the beginning and all the bits in the  $flag$  that is of size  $itemcount$  are set to 1 to indicate so. The  $jump$  variable is equalized to the shift value associated with the character that is the immediate successor of the current window, as this character will be included in the next search attempt for sure.

The window is traversed in the previously computed scan order. While passing over a character  $ch$  at position  $pos$ , the  $flag$  is updated via a bitwise *and* operation with  $MaskMatrix[ch][pos]$  and the  $jump$  becomes the maximum of itself and the  $ShiftMatrix[ch][pos]$ . If all the bits of the  $flag$  become 0 at any step, the inner *for* loop aborts, which means none of the patterns can exist in the investigated text factor. At least one of the patterns is found, when the  $flag$  is not set to zero until the end of the *for* loop. In this case, the bits that are equal to 1 in the  $flag$  actually represent the items from the alignment matrix that exist in the scanned region.

Let's search the sample patterns on  $T[0..11]=yağreçelbal$  with the algorithm. First  $T$  is padded with  $maxlen = 6$  null characters. Initially  $flag = 111111$  and  $jump = ShiftMatrix[e][6] = 2$ . According to the scan order,  $T[2]$  ( $\grave{g}$ ) is visited. Flag becomes 0001000 as a result of the *and* operation with  $MaskMatrix[\grave{g}][2] = 0001000$ , and  $jump = 3$ , as  $ShiftMatrix[\grave{g}][2] = 3$  is greater than its previous value 2. Flag is not zero yet, and we continue with  $T[5]$ . Again we update the  $flag$  and  $jump$  variables by using

```

Calculate AlignmentMatrix;
Calculate MaskMatrix;
Calculate ShiftMatrix;
Calculate ScanOrder;
pad the text with maxlen number of null characters;
i=0;
flag = bitstream of itemcount bits;
while (i<n){
  jump = ShiftMatrix[T[i+maxlen]][maxlen];
  set all bits of the flag to 1;
  for (j=0 ; flag & j<maxlen ; j++){
    flag &= MaskMatrix[T[i+ScanOrder[j]]][ScanOrder[j]];
    jump = max(jump,
      ShiftMatrix[T[i+ScanOrder[j]]][ScanOrder[j]]);
  }
  if (flag){
    Check bits of the flag to find out detected items;
  }
  i += jump;
}

```

Fig. 1. Main search loop of the *tara* algorithm.

$MaskMatrix[\zeta][5] = 0100111$  and  $ShiftMatrix[\zeta][5] = 2$ , which returns  $jump = 3$  and  $flag = 0000000$ .

As all bits in flag is zero meaning no patterns can exist in  $T[0 \dots 5]$ , the window is slid right by the  $jump = 3$  amount. Now the window is located on  $T[3 \dots 8]$ . The  $flag$  and  $jump$  variables processed in the same way by visiting the characters  $T[3+2]$  (ç),  $T[3+5]$  (b),  $T[3+4]$  (l),  $T[3+3]$  (e),  $T[3+1]$  (e),  $T[3+0]$  (r) in order. At the end, the  $flag = 0100000$  indicating that the 5<sup>th</sup> item of the alignment matrix (reçel) is detected. The  $jump$  value is 5, and  $i = 3 + 5 = 8$  for the next attempt.

The text portion under investigation is now  $T[8 \dots 13]$ . Note that  $T[11 \dots 15]$  are all null as a result of the padding performed at the beginning. By similar operations, the  $T[10]$  (l),  $T[13]$  (null),  $T[12]$  (null),  $T[11]$  (null),  $T[9]$  (a),  $T[8]$  (b) are visited. The  $flag$  becomes 0000001, which means the first item bal from the alignment matrix is observed within the investigated window. When the current value of  $jump = 7$  is added to  $i$ ,  $i$  becomes 15 that is greater than the text length 12. Hence, the search is completed.

### III. ANALYSIS OF THE ALGORITHM

The computation of *alignment*, *mask*, and *shift* matrices are of quadratic order in time with a total space requirement of  $O((maxlen \times itemcount) + (|\Sigma| \times maxlen) + (|\Sigma| \times (maxlen + 1)))$ . The *scan order* calculation is linear and requires  $O(m)$  space. Thus, the total time and space complexity of the preprocessing steps in *tara* algorithm are both quadratic.

In each attempt of scanning the window of size  $maxlen$ , at least  $\lfloor \frac{maxlen}{minlen} \rfloor$ , and at most  $maxlen$  characters are visited. The shift amount is at least 1 and at most  $maxlen + 1$ .

In worst case, the shift amount is always one and all the characters in the window are checked. Hence, the worst-case time complexity is  $O(n \times maxlen)$ .

Best case is achieved when minimum number of character are checked with a maximum amount of shift. The best-case time complexity is then  $O(\lceil \frac{n}{maxlen+1} \rceil \times \lfloor \frac{maxlen}{minlen} \rfloor)$ . If we assume  $maxlen = minlen$  for the best case, which means all the patterns in  $P$  are of same length, the complexity becomes  $O(\lceil \frac{n}{maxlen+1} \rceil)$ . In summary, worst and best cases are of quadratic and sublinear order respectively. The performance of the algorithm in practice is investigated by the experiments described in the next section.

In general, bit-parallel algorithms define limitations on the number or lengths of the patterns. For example, the multi-pattern matching *Shift-And* [11] algorithm restricts that the total length of the patterns in  $P$  should be less than the machine word size denoted by  $w$ . In practise *tara* algorithm also defines a restriction that the corresponding *itemcount* of input patterns should be less than  $w$ . Hence, maximum number of patterns that can be searched effectively by the algorithm is  $w$  when all the patterns are equal in length of any size. When compared with the limitation defined in *Shift-And* algorithm, this restriction is much more flexible as  $\sum_{i=0}^{i < m} lp_i \leq w$  defines a more tight bound than  $\sum_{i=0}^{i < m} maxlen - lp_i + 1 \leq w$ .

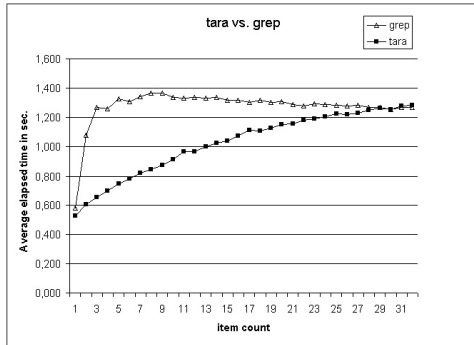
The algorithm scans the text in chunks of  $maxlen$  size windows. Thus, besides its bit-parallel characteristic, it also benefits from factor base scanning. Another compatible factor based multi-pattern matching algorithm was defined previously by Wu-Manber [13]. The shift amount in that algorithm cannot exceed the  $minlen$  size in case of a mismatch as also indicated by the authors in their study. On the other side, the new algorithm is capable of making  $maxlen + 1$  jumps, which is the maximum possible safe shift theoretically.

*Tara* algorithm is capable of handling character classes and bounded gaps in patterns as well. The inclusion of character classes is depicted before in the example on which the algorithm is described. For the bounded gaps, it is assumed that all characters are possible for the positions marked as don't cares. As an example, when the third character on word *bi.inci* is marked as don't care, the algorithm assumes any character of the alphabet appropriate for that position.

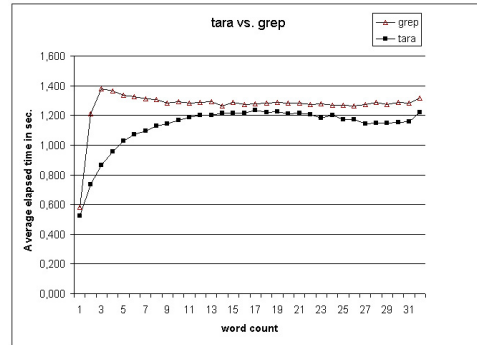
### IV. EXPERIMENTAL RESULTS

Most of the studies in the area have compared their performance against the *grep* function and also against the AC and CW variants. Hence, similar benchmarks are performed with the proposed algorithm.

Experiments are performed with a 1.8 Ghz machine with 512 MB memory running WindowsXP operating system on 100 MB of Turkish text that is collected from a daily newspaper. 50 words for each length of 2 to 20 are randomly chosen from the corpus. The selected  $50 \times 19 = 950$  words are again combined randomly to create the test pattern sets. While forming the pattern sets, it is checked that the corresponding *itemcount* of the set does not exceed the machine word size, which is 32 for the test computer. 8500 such sample

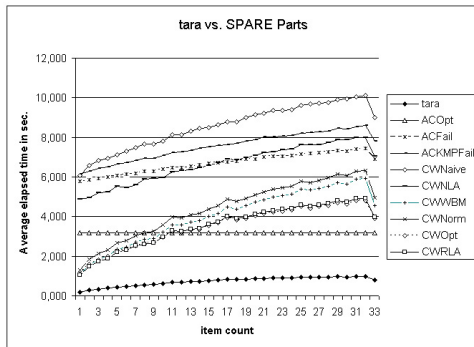


(a) According to *item count*

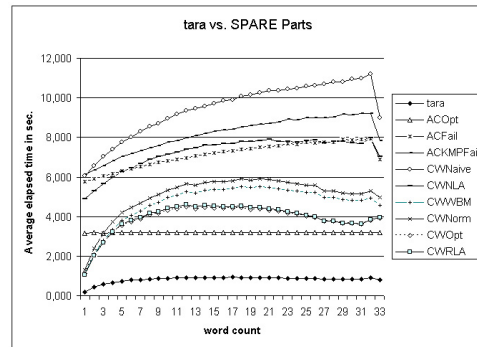


(b) According to *word count*

Fig. 2. The performance of *tara* versus *grep* according to itemcount and word count



(a) According to *item count*



(b) According to *word count*

Fig. 3. The performance of *tara* versus Aho-Corasick and Commentz-Walter variants via *SPARE Parts* toolkit

pattern sets, where the cardinalities vary from 2 to 32, are formed. Special attention is paid to generate approximately equal number of sets from each cardinality.

The performance of the new algorithm is first compared with the *GNU grep 2.5.1a* search utility. Elapsed time for searching each *tefig:mainloopst* pattern set by the *grep* and *tara* programs is recorded. The results are depicted in Figure 2. Although, the *itemcount* factor actually specifies the real performance of the *tara* algorithm, average timings are also grouped according to word count to be consistent with the literature on the topic.

The experiments show that the overall gain obtained is about 10%, and especially for pattern sets whose corresponding *itemcount*  $\leq 10$ , *tara* returns a speed up of 1.63 times. Up to 70% performance gain is observed on the test sets. When the *itemcount* comes close to 32, the performance of *tara* decreases as expected. That is because, each column of the alignment matrix contains more characters with increasing number of patterns, which lowers the shift amounts considerably.

During the benchmark of *tara* versus *grep*, both programs are called from system shell, and file I/O operations are included in the elapsed time calculations. It is observed that *tara* deals with file operations about half of the elapsed time. As *grep* has a more sophisticated implementation, the

performance is optimized in itself.

In the second part of the experiments, *tara* is compared against 9 variants of AC and CW included in *SPARE Parts* toolkit [15] that is being maintained since 1993. The authors of the toolkit indicate that although the version used in this study is optimized for safety, it does not sacrifice speed also. During this second benchmark, file I/O operations are excluded from the timings.

Three types of AC and six types of CW are included in the toolkit. The AC variants differ in the transition machines used. *ACOpt*, *ACFail*, and *ACKMPFFail* use optimal transition function, failure function with extended trie, and Knuth-Morris-Pratt [16] failure function with forward trie respectively. The CW type algorithms are formed by applying different shifters. The naive version with a shift of one symbol is named *CWNaive*. The standard and optimized versions are *CWNorm* and *CWOpt*. *CWRLA* and *CWNLA* implementations are with lookahead right one symbol and without using lookahead symbol respectively. The last one is *CWWBM* which includes the weak Boyer-Moore shift function. The average timings are depicted in Figure 3.

The results show that *tara* is the absolute winner among the 9 AC and CW variants with an overall speed up of approximately 3.5 times from its nearest successor. On the test pattern sets, where *itemcount*  $\leq 10$ , the performance gain is

5 times on the average.

## V. CONCLUSION

In this study, a new multiple pattern matching algorithm benefitting from bit-parallelism is introduced. Besides its quadratic worst case time complexity, it performs very fast on the average. The experimental results on language text indicate that, for small number of patterns the unoptimized implementation of the algorithm is approximately 1.5 times faster than *grep* software and 5 times than its nearest successor of the AC and CW variant.

The bottleneck of the algorithm is the restriction that the defined  $itemcount = \sum_{i=0}^{i < m} maxlen - lp_i + 1$  should be less than the machine word size. It is believed that for practical usage it represents a very convenient way of searching with its simplicity and speed as the machine word size of the computers today are sufficient for daily life problems. It worths here to note that the algorithm is very suitable for hardware implementation also. By this way the machine word size limit may be relaxed also.

## REFERENCES

- [1] C. Charras and T. Lecroq, *Handbook of exact string matching algorithms*. King's Collage Publications, 2004.
- [2] B. Watson, "Taxonomies and toolkits of regular language algorithms," Ph.D. dissertation, Faculty of Computer Science, Eindhoven University of Technology, The Netherlands, September 1995.
- [3] B. Watson and G. Zwaan, "A taxonomy of sublinear multiple keyword pattern matching algorithms," *Science of Computer Programming*, vol. 27, no. 2, pp. 85–118, 1996.
- [4] B. Watson, "The performance of single keyword and multiple keyword pattern matching algorithms," in *Proceedings of the Third South American Workshop on String Processing*, Recife, Brazil, August 1996.
- [5] A. Aho and M. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, vol. 18, pp. 333–340, June 1975.
- [6] B. Commentz-Walter, "A string matching algorithm fast on the average," in *Proceedings of 6th International Colloquium on Automata, Languages, and Programming*, 1979, pp. 118–132.
- [7] R. Boyer and J. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.
- [8] R. Baeza-Yates, "Improved string searching," *Software – Practice and Experience*, vol. 19, pp. 257–271, 1989.
- [9] N. Horspool, "Practical fast searching in strings," *Software – Practice and Experience*, vol. 10, 1980.
- [10] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter, "Fast practical multi-pattern matching," *Information Processing Letters*, vol. 71, no. 3–4, pp. 107–113, 1993.
- [11] R. Baeza-Yates and G. Gonnet, "A new approach to text searching," *Communications of the ACM*, vol. 35, no. 10, pp. 74–82, 1992.
- [12] S. Kim and Y. Kim, "A fast multiple string-pattern matching algorithm," in *Proceedings of 17th AoM/IAoM Conference on Computer Science*, August 1999.
- [13] S. Wu and U. Manber, "A fast algorithm for multi-pattern matching," The Computer Science Department, The University of Arizona, Tech. Rep., 1994.
- [14] D. Sunday, "A very fast substring search algorithm," *Communications of the ACM*, vol. 33, no. 8, pp. 132–142, 1990.
- [15] B. Watson and L. Cleophas, "SPARE parts: A C++ toolkit for string pattern recognition," *Software – Practice and Experience*, vol. 34, pp. 697–710, 2004.
- [16] D. Knuth, J. Morris, and V. Pratt, "Fast pattern matching in strings," *SIAM Journal of Computing*, vol. 6, no. 2, pp. 323–350, 1977.