Two-Dimensional Bitwise Memory Matrix: A Tool for Optimal Parallel Approximate Pattern Matching*

Jan Šupol and Bořivoj Melichar

Department of Computer Science & Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo nám. 13, 121 35 Prague 2
jan.supol@gmail.com, melichar@fel.cvut.cz

Abstract. A very fast parallel approach to pattern matching is presented. The approach is based on the bit-parallel approach and we use two-dimensional bitwise memory matrix which helps to achieve very fast parallel pattern matching algorithms. The parallel pattern matching takes $\mathcal{O}(1)$ time for the exact pattern matching and $\mathcal{O}(k)$ for the approximate pattern matching, where k is the number of errors.

1 Introduction

The pattern matching problem is to find all occurrences of a given pattern $P = p_1p_2 \dots p_m$ in a larger text $T = t_1t_2 \dots t_n$, $n > m$, both sequences of symbols from a given alphabet $A = a_1a_2 \dots a_{|A|}$.

Many different solutions of this problems are known, and we are interested in the pattern matching using finite nondeterministic automata. A finite automaton (*FA*) is a quintuple (Q, A, δ, I, F) where Q is a finite set of states, A is a finite input alphabet, and $F \subseteq Q$ is a set of final states. If *FA* is nondeterministic (*NFA*), then δ is a mapping $Q \times (A \cup \{\varepsilon\}) \mapsto P(Q)$ and $I \subseteq Q$ is a set of initial states. If *FA*$= (Q, A, \delta, q_0, F)$ is deterministic (*DFA*), then δ is a (partial) function $Q \times A \mapsto Q$ and q_0 is the only initial state. We refer to *NFA* used for pattern matching as a pattern matching automaton (*PMA*).

Hamming distance $H(x, y) \leq k$ is maximum k substitutions (replace operations) required to transform string x into string y (see [4]). *Levenshtein distance* $L(x, y) \leq k$ is maximum k operations replace, insert, or delete required to transform string x into string y. *PMA* for pattern P using the Hamming distance k is a pattern matching automaton that matches any pattern X, such that $H(P, X) \leq k$. *PMA* for pattern P using the Levenshtein distance k is a pattern matching automaton that matches any pattern X, such that $L(P, X) \leq k$.

The running of *PMA* can be simulated by the bit-parallel algorithms. This technique was introduced in [2] ("shift-and" variation), and it was improved in [1, 9] ("shift-or" variation used in this paper). It has been shown [5], that the bit-parallel algorithms simulates *NFA* and we use these algorithms for the parallel pattern matching.

* This research has been partially supported by the Ministry of Education, Youth, and Sport of the Czech Republic under research program MSM6840770014, by the Czech Science Foundation as project No. 201/06/1039, and by the Czech Technical University as project No. CTU0609613

Parallel pattern matching was quite a popular topic. We might cite a constant time string matching algorithm [3], which has the same time complexity as our algorithm for the exact pattern matching, though it does not use the bit-parallelism. We might also cite an $\mathcal{O}(log\ m + k)$ time [8], or an $\mathcal{O}(k)$ time [7] algorithms for the approximate pattern matching. These have similar time complexity to our $\mathcal{O}(k)$ algorithm for approximate pattern matching, but our algorithm needs a smaller number of processors.

Our algorithm needs *EREW PRAM* for the exact string matching and *CREW PRAM* for the approximate string matching. More information for the parallel computation models is e.g. in [6]. We also need a shared memory organized as a matrix of size $\mathcal{O}(n \times n)$ bits, where n is the length of the text. Since bit-parallel algorithms work with a computer word of length m, where m is the length of the pattern, we need to access a whole word in the bit-memory matrix. Here we have a strong condition. We need to access this memory both on rows and on columns. Therefore we use two operations to access the memory matrix. The first is $MEMX[index_x][index_y]$ accessing a word in a column with index $index_x$ starting with the bit on a row with index $index_y$ and the second is $MEMY[index_x][index_y]$ accessing a word on a row with index $index_y$ starting with the bit in a column with index $index_x$. This accessibility is enough to present a cost-optimal parallel approximate pattern matching algorithm.

A parallel algorithm is cost-optimal if its time processor product is equal to the time of the best known sequential algorithm solving the same issue.

We use some bitwise operations in this paper. Operation **or** is a standard bitwise OR operation and operation **and** is a standard bitwise AND operation. Operation **shl** is a standard shift-left bitwise operation, and the right-most bit is set to 0. Operation **shr** is a standard shift-right bitwise operation, and the left-most bit is set to 1. We also use operation $\mathbf{shl}^i(x)$ as the operation **shl** performed i times on bit-vector x.

This paper is organized as follows. Section 2 explains the "shift-or" variation of a bit-parallel algorithm. Section 3 discuss the parallel variation of thr "shift-or" algorithm. Section 4 provides a conclusion.

2 Bit-Parallelism

Here we explain the "shift-or" variation of the bit-parallel algorithm. It uses matrices $R^l, 0 \leq l \leq k$ of size $m \times (n+1)$, and matrix D of size $m \times |A|$, where k is the maximum number of edit operations in pattern P. Each element $r^l_{j,i}, 0 \leq i \leq n$ contains 0, if the edit distance between string $p_1 p_2 \ldots p_j$ and the string ending at position i in text T is $\leq l$, or 1, otherwise. Each element $d_{j,x}, 0 < j \leq m$, $x \in A$, contains 0, if $p_j = x$, or 1, otherwise.

In exact string matching, vectors $R^0_i, 0 \leq i \leq n$, are computed as follows:

$$\begin{array}{ll} r^0_{j,0} = 1, & 0 < j \leq m \\ R^0_i = \mathbf{shl}(R^0_{i-1})\ \mathbf{or}\ D[t_i], & 0 < i \leq n \end{array} \tag{1}$$

In approximate string matching using the Hamming distance, vectors $R^l_i, 0 \leq l \leq k, 0 \leq i \leq n$, are computed as follows:

$$\begin{array}{ll} r^l_{j,0} = 1, & 0 < j \leq m, 0 \leq l \leq k \\ R^0_i = \mathbf{shl}(R^0_{i-1})\ \mathbf{or}\ D[t_i], & 0 < i \leq n \\ R^l_i = (\mathbf{shl}(R^0_{i-1})\ \mathbf{or}\ D[t_i])\ \mathbf{and}\ \mathbf{shl}(R^{l-1}_{i-1}), & 0 < i \leq n, 0 < l \leq k \end{array} \tag{2}$$

In approximate pattern matching using the Levenshtein distance, vectors $R_i^l, 0 \leq l \leq k, 0 \leq i \leq n$, are computed as follows:

$$\begin{array}{ll} r_{j,0}^l = 0, & 0 < j \leq l, 0 < l \leq k \\ r_{j,0}^l = 1, & l < j \leq m, 0 \leq l \leq k \\ R_i^0 = \mathbf{shl}(R_{i-1}^0) \textbf{ or } D[t_i], & 0 < i \leq n \\ R_i^l = (\mathbf{shl}(R_{i-1}^l) \textbf{ or } D[t_i]) & \\ \quad \textbf{and } \mathbf{shl}(R_{i-1}^{l-1} \textbf{ and } R_i^{l-1}) & \\ \quad \textbf{and } (R_{i-1}^{l-1} \textbf{ or } V), & 0 < i \leq n, 0 < l \leq k \end{array} \tag{3}$$

The auxiliary vector V is computed as follows:

$$V = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_m \end{bmatrix}, \text{ where } v_m = 1 \text{ and } v_j = 0, \forall j, 1 \leq j < m. \tag{4}$$

The term $\mathbf{shl}(R_{i-1}^l)$ **or** $D[t_i])$ represents matching – position i in text T is increased, the position in pattern P is increased by operation **shl**, and the positions corresponding to the input symbol t_i are selected by term **or** $D[t_i]$. The term $\mathbf{shl}(R_{i-1}^{l-1})$ represents edit operation replace – position i in text T is increased, the position in pattern P is increased, and edit distance l is increased. The term $\mathbf{shl}(R_i^{l-1})$ represents edit operation delete – the position in pattern is increased, the position in the text is not increased, and edit distance l is increased. The term R_{i-1}^{l-1} represents edit operation insert – the position in the pattern is not increased, the position in the text is increased, and edit distance l is increased. The term **or** V provides that no insert transition leads from any final state.

D	a	b	c	d	$A \setminus \{a, b, c, d\}$
a	0	1	1	1	1
d	1	1	1	0	1
b	1	0	1	1	1
b	1	0	1	1	1
c	1	1	0	1	1
a	0	1	1	1	1

Table 1. Matrix D for the pattern $P = adbbca$

An example of mask matrix D for the pattern $P = adbbca$ is shown in Table 1 and an example of matrix R^0 for exact pattern matching and matrix R^1 for approximate string matching using the Levenshtein distance $k = 2$, respectively, is shown in Table 2.

3 Parallelization of the bit-parallel algorithms

This section focuses on bit-parallel simulation of the nondeterministic *PMA*. The motivation is to use many processors, each with computer words long enough to fit into a single register bit-vector m bits in length, and to make the bit-parallel algorithms truly parallel.

R^0	–	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	0	1	1	0	1	1	0	0	1	0	1	1	1	1	0
d	1	1	0	1	1	1	1	1	1	1	1	0	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
R^1	–	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
d	1	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0
b	1	1	0	0	1	0	1	1	1	0	1	0	0	0	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
R^2	–	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
d	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0
c	1	1	1	0	1	1	0	1	1	1	1	1	0	0	0	0
a	1	1	1	1	0	1	1	0	1	1	1	1	1	0	0	0

Table 2. Matrices R^0, R^1, and R^2 for pattern matching using the Levenshtein distance, $T = adcabcaabadbbca$, $k = 2$, $P = adbbca$

Now we explain the idea originating in Formula (1) using the "shift-or" algorithm. The first bit-vectors may be computed as follows:

$$
\begin{aligned}
R_0^0 &= R_0^0 \\
R_1^0 &= \mathbf{shl}(R_0^0)\ \mathbf{or}\ D[t_1] \\
R_2^0 &= \mathbf{shl}(R_1^0)\ \mathbf{or}\ D[t_2] = \mathbf{shl}(\mathbf{shl}(R_0^0)\ \mathbf{or}\ D[t_1])\ \mathbf{or}\ D[t_2] = \\
&= \mathbf{shl}^2(R_0^0)\ \mathbf{or}\ \mathbf{shl}(D[t_1])\ \mathbf{or}\ D[t_2] \\
R_3^0 &= \mathbf{shl}(R_2^0)\ \mathbf{or}\ D[t_3] = \mathbf{shl}(\mathbf{shl}(R_1^0)\ \mathbf{or}\ D[t_2])\ \mathbf{or}\ D[t_3] = \\
&= \mathbf{shl}(\mathbf{shl}(\mathbf{shl}(R_0^0)\ \mathbf{or}\ D[t_1])\ \mathbf{or}(D[t_2]))\ \mathbf{or}\ D[t_3] = \\
&= \mathbf{shl}^3(R_0^0)\ \mathbf{or}\ \mathbf{shl}^2(D[t_1])\ \mathbf{or}\ \mathbf{shl}(D[t_2])\ \mathbf{or}\ D[t_3]
\end{aligned}
$$

Hence the following equation might be proven:

$$R_i^0 = \mathbf{shl}^i(R_0^0)\ \mathbf{or}\ \mathbf{shl}^{i-1}(D[t_1])\ \ldots\ \mathbf{or}\ \mathbf{shl}(D[t_{i-1}])\ \mathbf{or}\ D[t_i],\ 1 \le i \le n \qquad (5)$$

Using Formula (5) we may compute the example for the pattern $P = adbbca$ and text $T = adcbcadbbca$ depicted in Table 3. Note that we use the same matrix D as in Section 2, Table 1.

The initial bit-vector, left-shifted by $i = n = 11$ bits, is in the first column labeled with the symbol "-". In each other column, except the last one, there is a bit-vector from matrix D left-shifted by some number of bits. The **or** operation between all these rows gives the result, and as we can see, a match occurs only in one position, the same as in Table 2. We may conclude many interesting observations from this example.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
R^0	$-$	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a	OR
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	1
2	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	1
3	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	1
4	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**0**	**1**	**1**	1
5	0	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	0
6	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**0**	**1**	**1**	1	1
7	0	0	0	0	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1
8	0	0	0	0	0	0	0	**0**	**1**	**0**	**1**	**1**	**1**	1	1	1	1
9	0	0	0	0	0	0	**1**	**1**	**1**	**0**	**1**	**1**	1	1	1	1	1
10	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**0**	1	1	1	1	1	1
11	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1
12	0	0	0	**1**	**1**	**0**	**1**	**1**	**0**	1	1	1	1	1	1	1	1
13	0	0	**1**	**1**	**1**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	1
14	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1
15	**1**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	1
16	**1**	**1**	**1**	**0**	**0**	1	1	1	1	1	1	1	1	1	1	1	1
17	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	1	1	1
18	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	1	1	1	1
19	**1**	**0**	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
20	**1**	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 3. Matrix R^0 for pattern matching for the exact pattern, $P = adbbca$, $T = adcabcaabadbbca$

There are only two reasons for the initial bit-vector R_0^0 to have something to start sequentially with, and to disallow a match on the prefix of the text T shorter than pattern P.

The only interesting part of the matrix is the bold part. We can also exclude the first $m-1=5$ rows labeled with symbols "a, d, b, b, c" and the last $m=6$ rows, since there cannot be a match (in the exact case). Also the initial bit-vector is not important.

We implement the shift by j-bits $\mathbf{shl}^j(D[t_i])$ operation as a writing of one computer word m-bits long into a memory in a specified position using operation $MEMX[n-j][j] \leftarrow D[t_i]$. Having this memory organization, there is a word on each row of the memory $MEMY[n-j][j]$ (a bold one in Table 3), which contains a crucial information:

Proposition 1. *The word in memory $MEMY[n-j+1][j-1] > 0, 1 \leq j \leq n$ if and only if $t_{n-j+1}t_{n-j+2}\ldots t_{n-j+m} \neq p_1p_2\ldots p_m$.*

Proof. Simply from the definition of matrix D. If $t_{n-j+i} = p_i$ then $d_{i,n-j+i} = 0$ or 1 otherwise, $1 \leq j \leq n, 1 \leq i \leq m$. But each bit $d_{i,n-j+i}$ is in the $(j-1)$-th row, because vector $D[t_{n-j+i}]$ has been shifted by $j-i$ bits. □

The **or** operation is then a comparison, whether a computer word in a row is zero (a match), or non-zero (a mismatch) and we may compute:

$$R_i^0 = MEMY[n-i+1][i-1], \qquad m \leq i \leq n \tag{6}$$

We are not interested in the bit-vectors $R_i^0, 0 < i < m-1$, because there can not be a match.

If the access time to the memory using both *MEMX* and *MEMY* operations is $\mathcal{O}(1)$, string-matching in parallel takes only $\mathcal{O}(1)$ parallel time using n processors.

3.1 Parallel pattern matching using the Hamming distance

The observation of Table 3 may continue. Matrix R^l no longer consists only of bit-vectors $R^l_i, 0 \leq l \leq k, 0 \leq i \leq n$.

The bit-vectors R^0_i computed by Formula 1 in the sequential algorithm on the diagonal of the matrix were composed of the bit-vectors $n-i+1, n-i+2, \ldots, n-i+m$. However, they are slightly different: the bit-vectors computed in parallel contain more bits set to zeros, more active states, because they were not deactivated using prefix computation by the operation **or**. In exact pattern matching this was not important, because Proposition 1 ensures at least one bit set to one if there is no match.

The advantage is that the number of bits set to 1 indicates the number of replace operations needed for a matching.

However, using the Hamming distance this difference is most important. Since we are going to perform "replace" operations in parallel, we could perform more than one "replace", each in a different position and we cannot guarantee the number of these "replace" operations. Therefore we need vectors R^0_j exactly as in the sequential version. This ensures only one "replace" operation performed in each of k levels, because the "replace" is only after a "match", except the first symbol.

C^0	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a	OR
5	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	0
6	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	1	1
7	0	0	0	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1
8	0	0	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1
9	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1
10	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1
11	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1
12	0	0	**1**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1
13	0	**1**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1
14	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1
R^1	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a	OR
5	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	0
6	0	0	0	0	0	0	0	0	**0**	**1**	**1**	**0**	**1**	**1**	1	1
7	0	0	0	0	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1
8	0	0	0	0	0	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1
9	0	0	0	0	0	**0**	**1**	**1**	**0**	**1**	**1**	1	1	1	1	1
10	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**0**	1	1	1	1	1	1
11	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1
12	0	0	**0**	**1**	**0**	**1**	**1**	**0**	1	1	1	1	1	1	1	1
13	0	**0**	**1**	**1**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	1
14	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1

Table 4. Matrices C^0, and R^1 for pattern matching using the Hamming distance, $T = adcbcadbbca$, $k = 3$, $P = adbbca$

It is very easy to find the first (highest, left-most) bit set to 1 and to set all lower bits to 1 as well. This operation can be performed as:

$$R^0_i \leftarrow R^0_i \textbf{ or } 2^{\lceil log_2 R^0_i \rceil}, \quad m \leq i \leq n \tag{7}$$

C^1	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a	OR
5	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	0
6	0	0	0	0	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1
7	0	0	0	0	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1
8	0	0	0	0	0	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1
9	0	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1
10	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1
11	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1
12	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1
13	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1
14	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1
R^2	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a	OR
5	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	0
6	0	0	0	0	0	0	0	0	**0**	**0**	**1**	**0**	**1**	**1**	1	1
7	0	0	0	0	0	0	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1
8	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**1**	**1**	1	1	1	1
9	0	0	0	0	0	**0**	**0**	**1**	**0**	**1**	**1**	1	1	1	1	1
10	0	0	0	0	**0**	**0**	**1**	**1**	**1**	**0**	1	1	1	1	1	1
11	0	0	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	1	1
12	0	0	**0**	**0**	**0**	**1**	**1**	**0**	1	1	1	1	1	1	1	1
13	0	**0**	**0**	**1**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	1
14	**0**	**0**	**0**	**0**	**1**	**1**	1	1	1	1	1	1	1	1	1	1

Table 5. Matrices C^1, and R^2 for pattern matching using the Hamming distance, $T = adcbcadbbca$, $k = 3$, $P = adbbca$

This computation is very fast, $\mathcal{O}(1)$ time. For example the operation $log_2 x$ is computed using instruction *FYL2X* on X86 processors.

We use the observations on this section and we can formulate the idea of parallel pattern matching using the Hamming distance, which could be used later with the Levenshtein distance.

We compute the corrected matrix $R^l, 0 \leq l \leq k$ from each matrix R^l using Formula 7. We refer to this corrected matrix as C^l. The shifted vectors $D[t_i], 0 \leq i \leq n$ placed in matrix R^l are no longer in matrix C^l. Thus we refer to these vectors (shown in bold in Table 3) as C_i^l.

The observation of matrix R^0 revealed that each vector $R_l^0, m \leq l \leq n$ contains the bits set to 1 if a replace operation is needed. Each vector R_i^l in matrix C^l refers to a prefix successfully matched with at most l substitutions and in matrix R^{l+1} we may add one substitution more. Thus we compute each matrix $R^l, 0 < l \leq k$ as follows:

$$\begin{aligned} R_i^l = {} & \mathbf{shl}^i(R_0^l) \textbf{ or } (\mathbf{shl}^{i-1}(D[t_1]) \textbf{ and } C_0^{l-1}) \ \ldots \\ & \textbf{or } (\mathbf{shl}(D[t_{i-1}]) \textbf{ and } C_{i-2}^{l-1}) \\ & \textbf{or } (D[t_i] \textbf{ and } C_{i-1}^{l-1}),\ 1 \leq i \leq n,\ 1 \leq l \leq k \end{aligned} \tag{8}$$

An example of parallel pattern matching using the Hamming distance for the pattern $P = adbbca$, $k = 3$, $T = adcbcadbbca$ is given in Table 4, Table 5, and Table 6, respectively. The first table contains matrix C^0, which has been corrected by Formula 7 from matrix R^0, and matrix R^1, computed by Formula 8. The second table contains matrix C^1 corrected by Formula 7 from matrix R^1, and matrix R^2 computed by Formula 8. The last table contains matrix C^2 corrected by Formula 7 from matrix

C^2	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a	OR
5	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	0
6	0	0	0	0	0	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1
7	0	0	0	0	0	0	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1
8	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**1**	**1**	1	1	1	1
9	0	0	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1
10	0	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1	1
11	0	0	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	1	1
12	0	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	1	1	1
13	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1
14	**0**	**0**	**0**	**0**	**1**	**1**	1	1	1	1	1	1	1	1	1	1
R^3	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a	OR
5	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	0
6	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**1**	**1**	1	1
7	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**1**	**1**	1	1	1
8	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**1**	1	1	1	1
9	0	0	0	0	0	**0**	**0**	**0**	**0**	**1**	**1**	1	1	1	1	1
10	0	0	0	0	**0**	**0**	**0**	**1**	**1**	**0**	1	1	1	1	1	1
11	0	0	0	**0**	**0**	**0**	**0**	**1**	**1**	1	1	1	1	1	1	1
12	0	0	**0**	**0**	**0**	**0**	**1**	**0**	1	1	1	1	1	1	1	1
13	0	**0**	**0**	**0**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	0
14	**0**	**0**	**0**	**0**	**0**	**1**	1	1	1	1	1	1	1	1	1	1

Table 6. Matrices C^2, and R^3 for pattern matching using the Hamming distance, $T = adcbcadbbca$, $k = 3$, $P = adbbca$

R^2, and matrix R^3 computed by Formula 8. These tables are shortened, as described above.

3.2 Parallel pattern matching using the Levenshtein distance

Parallel pattern matching using the Hamming distance is very similar to the pattern matching using the Levenshtein distance, though in addition we must consider the operations "insert" and "delete".

Recall Formula 3. We computed $R_i^l, 0 < i \leq n, 0 < l \leq k$ in the sequential algorithm as:

$$\begin{aligned} R_i^l = &(\mathbf{shl}(R_{i-1}^l) \textbf{ or } D[t_i]) \\ &\textbf{and } \mathbf{shl}(R_{i-1}^{l-1} \textbf{ and } R_i^{l-1}) \\ &\textbf{and } (R_{i-1}^{l-1} \textbf{ or } V), \qquad 0 < i \leq n, 0 < l \leq k \end{aligned} \tag{9}$$

We use exactly the same logic rules for operations "insert" and "delete", and we may formulate for the Levenshtein distance:

$$\begin{aligned} R_i^l = &\mathbf{shl}^i(R_0^l) \\ &\textbf{or } (\mathbf{shl}^{i-1}(D[t_1]) \textbf{ and } C_0^{l-1} \textbf{ and } \mathbf{shl}(C_1^{l-1}) \textbf{ and } (\mathbf{shr}(C_0^{l-1}) \textbf{ or } V)) \\ &\dots \\ &\textbf{or } (\mathbf{shl}(D[t_{i-1}] \textbf{ and } C_{i-2}^{l-1} \textbf{ and } \mathbf{shl}(C_{i-1}^{l-1}) \textbf{ and } (\mathbf{shr}(C_{i-2}^{l-1}) \textbf{ or } V)) \\ &\textbf{or } (D[t_i] \textbf{ and } C_{i-1}^{l-1} \textbf{ and } \mathbf{shl}(C_i^{l-1}) \textbf{ and } (\mathbf{shr}(C_{i-1}^{l-1}) \textbf{ or } V)), \\ &1 \leq i \leq n,\ 1 \leq l \leq k \end{aligned} \tag{10}$$

When computing vector R_i^l, each term C_{i-1}^{l-1} represents operation "replace" as when using the Hamming distance. Since this bit-vector has been already shifted by

C^0	–	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a	OR
3	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	
4	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	
5	0	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	0
6	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	1	1
7	0	0	0	0	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1
8	0	0	0	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1
9	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1
10	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1
11	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1
12	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1
13	0	0	**1**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1
14	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1
15	**1**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	1
16	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	1	1

R^1	–	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a	OR
3	0	0	0	0	0	0	0	0	0	0	0	0	**0**	**1**	**1**	**1**	
4	0	0	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	
5	0	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	0
6	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	1	0
7	0	0	0	0	0	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1
8	0	0	0	0	0	0	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1
9	0	0	0	0	0	0	**0**	**0**	**1**	**0**	**1**	**1**	1	1	1	1	1
10	0	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**0**	1	1	1	1	1	1
11	0	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1
12	0	0	0	**0**	**0**	**0**	**1**	**1**	**0**	1	1	1	1	1	1	1	1
13	0	0	**0**	**0**	**1**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	1
14	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1
15	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	1
16	**1**	**1**	**1**	**0**	**0**	1	1	1	1	1	1	1	1	1	1	1	1

Table 7. Matrices C^0 and R^1 for pattern matching using the Levenshtein distance, $T = adcbcadbbca$, $k = 2$, $P = adbbca$

$n - i + 1$ positions once, that is one more than $n - i$ shifts when computing the i-th column, thus no further shift operation is needed to transform the sequential term $\mathbf{shl}(R_{i-1}^{l-1})$ into a parallel term.

The term $(\mathbf{shr}(C_{i-1}^{l-1})$ **or** $V)$ represents the operation "insert". This vector has been shifted too much by the same logic as the bit-vector for operation "replace". Therefore we need to shift it one bit back, when transforming the sequential "insert" term $(R_{i-1}^{l-1}$ **or** $V)$ into a parallel term.

The term $\mathbf{shl}(C_i^{l-1})$ represents the operation "delete". The bit-vector C_i^{l-1} has been shifted exactly enough to shift it once more for the same reason as in the sequential algorithm.

Due to the operations "delete" and "insert" we need to shorten the original matrix R^0 less than when using the Hamming distance. We need the initial bit-vector R_0^l defined by Formula 3, which sets more states initial because of the "delete" ε-transitions from the initial state. We also need k rows before the $(m-1)$-th row for the operation "delete" and we need k rows after the $(n-1)$-th row for operation "insert".

An example of parallel pattern matching using the Levenshtein distance for the pattern $P = adbbca$, text $T = adcbcadbbca$ and $k = 2$ is given in Table 7 and Table

C^1	–	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a	OR
3	0	0	0	0	0	0	0	0	0	0	0	0	**0**	**1**	**1**	**1**	
4	0	0	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	
5	0	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	0
6	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	1	0
7	0	0	0	0	0	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1
8	0	0	0	0	0	0	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1
9	0	0	0	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1
10	0	0	0	0	0	**0**	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1
11	0	0	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1
12	0	0	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	1	1	1
13	0	0	**0**	**0**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1
14	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1
15	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	1
16	**1**	**1**	**1**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	1	1

R^2	–	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a	OR
3	0	0	0	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	
4	0	0	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	
5	0	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	0
6	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	1	0
7	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	1	1	0
8	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**1**	**1**	1	1	1	1
9	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**1**	**1**	1	1	1	1	1
10	0	0	0	0	0	**0**	**0**	**0**	**1**	**1**	**0**	1	1	1	1	1	1
11	0	0	0	0	**0**	**0**	**0**	**1**	**1**	**1**	1	1	1	1	1	1	1
12	0	0	0	**0**	**0**	**0**	**0**	**1**	**0**	1	1	1	1	1	1	1	1
13	0	0	**0**	**0**	**0**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	0
14	0	**0**	**0**	**0**	**0**	**1**	**1**	1	1	1	1	1	1	1	1	1	1
15	**0**	**0**	**0**	**0**	**1**	**1**	1	1	1	1	1	1	1	1	1	1	1
16	**0**	**0**	**0**	**0**	**0**	1	1	1	1	1	1	1	1	1	1	1	0

Table 8. Matrices C^1 and R^2 for pattern matching using the Levenshtein distance, $T = adcbcadbbca$, $k = 2$, $P = adbbca$

8. The former table contains matrix C^0 computed by Formula 7 from matrix R^0 in Table 3 and it also contains matrix R^1 computed from matrix C^0 by Formula 10. The latter contains matrix C^1 computed from matrix R^1 by Formula 7 and it also contains matrix R^2 computed from matrix C^1 by Formula 10.

4 Conclusion

We have presented the idea of parallel pattern matching using bit-parallelism (the "shift-or" variation).

We used a two-dimensional memory matrix which enabled very fast parallel pattern matching. Parallel pattern matching for an exact pattern takes $\mathcal{O}(1)$ parallel time, using n processors. The algorithm does not need any concurrent read or write operation, thus it can be implemented on *EREW PRAM* with shared two-dimensional memory. Since the processor-time product is $\mathcal{O}(n)$, it is a cost-optimal algorithm.

Parallel pattern matching using the Hamming distance with k substitutions takes $\mathcal{O}(k)$ parallel time, using n processors. The algorithm also does not need any concurrent read or write operation, thus it can also be implemented on *EREW PRAM*

with shared two-dimensional memory. The processor-time product is equal to the sequential time $\mathcal{O}(kn)$, hence this algorithm is also cost-optimal.

Parallel pattern matching using the Levenshtein distance with k substitutions takes $\mathcal{O}(k)$ parallel time, using $max(n+1, n-m+2k+1)$ processors. The algorithm needs a concurrent read operation when reading the same bit-vector $C_i^l, 0 \leq l < k, 1 \leq i \leq n-1$, thus it can be implemented on *CREW PRAM* with shared two-dimensional memory. The processor-time product is also equal to the sequential time $\mathcal{O}(kn)$.

Known parallel pattern matching algorithms are derived from dynamic programming, which takes $\mathcal{O}(mn)$ sequential time and therefore these parallel pattern matching algorithms are non-optimal. Parallel pattern matching derived from the bit-parallel algorithms can provide an optimal parallel pattern matching algorithm even when not using two-dimensional memory based on some ideas mentioned in this paper.

References

[1] R. A. Baeza-Yates and G. H. Gonnet: *A new approach to text searching.* Commun. ACM, 35(10) 1992, pp. 74–82.

[2] B. Dömölki: *An algorithm for syntactic analysis.* Computational Linguistics, 8 1964, pp. 29–46.

[3] Z. Galil: *A constant-time optimal parallel string-matching algorithm*, in Proceedings of the twenty-fourth annual ACM symposium on Theory of Computing, ACM Press, 1992, pp. 69–76.

[4] R. W. Hamming: *Coding and information theory (2nd ed.)*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[5] J. Holub: *Simulation of Nondeterministic Finite Automata in Pattern Matching*, Ph.D. Thesis, Czech Technical University in Prague, Feb. 2000.

[6] J. JáJá: *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.

[7] Y. Jiang and A. H. Wright: *O(k) parallel algorithms for approximate string matching.* Neural, Parallel and Scientific Computations, 1 1993, pp. 443–452.

[8] G. M. Landau and U. Vishkin: *Fast parallel and serial approximate string matching.* Journal of Algorithms, 10(2) 1989, pp. 157–169.

[9] S. Wu and U. Manber: *Fast text searching allowing errors.* Commun. ACM, 35(10) 1992, pp. 83–91.