Efficiency of Fast Parallel Pattern-Searching in Highly Compressed Texts

Leszek Gąsieniec* **Alan Gibbons*** **Wojciech Rytter *†**

Abstract

We consider efficiency of NC-algorithms for pattern-searching in highly compressed one- and two-dimensional texts. *"Highly compressed"* means that the text can be exponentially large with respect to its compressed version, and *"fast"* means *"in polylogarithmic time"*. Given an uncompressed pattern P and a compressed version of a text T, the *compressed matching* problem is to test if P occurs in T. Two types of closely related compressed representations of 1-dimensional texts are considered: the Lempel-Ziv encodings (LZ, in short) and restricted LZ encodings (RLZ, in short). For highly compressed texts there is a small difference between them, in extreme situations both of them compress text exponentially, e.g. Fibonacci words of size N have compressed versions of size $O(\log N)$ for LZ and Restricted LZ encodings. An efficient sequential algorithm for LZ-compressed matching was given in [7], we show that this algorithm is inherently sequential. Despite similarities we prove that LZ-compressed matching is P-complete while RLZ-compressed matching is rather trivially in NC. However a naive straightforward NC algorithm for RLZ-compressed matching needs too large (though polynomial) number of processors. We show how to perform RLZ-compressed matching efficiently using suffix trees and tree-contraction techniques applied to directed acyclic graphs with polynomial tree-size. Consequently we design a simple NC algorithm for RLZ-compressed matching which uses only a linear number of processors.

LZW-encoding is a special case (low compression, but more practical) of RLZ-encoding, and as a corollary we obtain a simple almost optimal parallel algorithm for LZW-compressed matching, which parallelizes efficiently the sequential almost linear time algorithm of [2].

We also consider the pattern-matching problem for highly compressed 2-dimensional texts, where we show how tree-contraction techniques can be used again to improve upon a naive NC-algorithm.

1 Introduction

In this paper we consider algorithms dealing with manipulations of *highly compressed* one- and two-dimensional texts (two dimensional arrays filled with symbols from a finite alphabet). The objects considered are potentially exponentially compressed. Denote by $\mathcal{T}$ a *compressed version* of a one- or two-dimensional text T. The *compressed pattern matching* problem for one- and two-dimensional texts is

Input: given *explicitly* pattern P and a compressed version $\mathcal{T}$ of T,
Output: *"yes"* if P occurs in T, otherwise *"not"*.

*Computer Science Dept., Liverpool University, Chadwick Building, Peach Street, Liverpool L69 7ZF, U.K.
†Institute of Informatics, Warsaw University, ul. Banacha 2, 02-097 Warszawa, Poland

The main size of the problem is the size $n = |\mathcal{T}|$ of the description of the text T. Denote $|P| = m$. Our model of computations is a *CRCW PRAM* , see [14].

The family of Lempel-Ziv encodings is one of the most successful compression techniques for 1-dimensional texts. We consider what is known as LZ1 or LZ77 encoding and its variations.

There are many different variations of LZ-encodings, we use the one based on factorizations of texts, similarly as in [7]. In LZ-encoding a given text w is represented as a composition of its subword-components (called here *factors*):

$$w = f_1\ f_2\ f_3\ \dots f_k$$

In LZ-encodings each f_i is a subword $w[p \dots q]$ of $f_1 f_2 \dots f_{i-1}$ or $f_i \in \Sigma$ and in the former case we identify f_i with the pair $[p, q]$. In the Restricted LZ-encoding each f_i is assumed to be a composition $f_p f_{p+1} \dots f_q$ of consecutive earlier factors, for $q < i$, or $f_i \in \Sigma$, in the former case we identify f_i with $[p, q]$. Σ is an input alphabet.

Observation 1. Observe that in LZ-encoding integers p, q correspond to positions in w and in RLZ-encoding they correspond to indices of factors, and are smaller. For highly compressed texts the numbers p, q can have n bits in LZ-encoding but only $\log n$ bits in RLZ-encoding. On the other hand RLZ encoding can have more factors.

Example. Define Fibonacci words by: $F_1 = b;\ F_2 = a;\ F_{k+2} = F_{k+1} \cdot F_k$, for $k \geq 0$

Then the following factorization of F_8 is both a LZ and a RLZ factorization:

$$Fib_8 = abaababaabaababaababa = f_1\ f_2\ f_3\ c_4\ f_5\ c_6\ f_7 = a\ b\ a\ aba\ baaba\ ababaaba\ ba$$

Hence an LZ-encoding of Fib_8 is

a b a [1,3] [2,6] [4,11] [2,3]

while a possible RLZ-encoding is

a b a [1,3] [2,4] [4,5] [2,3] .

It can happen that LZ factorization is not a RLZ one, for example LZ factorization

$$ababbaabb = a\ b\ ab\ ba\ abb$$

is not a Restricted LZ-factorization. We omit the proof of the following fact (point (1) has been essentially shown in [7] using idea of so called *fingers*).

Lemma 1.1
(1) *If n is the number of factors in a LZ-encoding of w then there is a RLZ-encoding of w with $O(n^2)$ factors.*
(2) *Assume $|\Sigma| = O(1)$ and w has a LZ-encoding of total size s, counting all bits, such that s is polylogarithmic with respect to $|w|$. Then there is a RLZ-encoding of w with total number of bits $O(s \cdot \log(s))$.*

Observation 2. In the factorizations we do not assume that the next factor is the longest possible, there are possibly many different RLZ-encodings of the same text. LZW-encodings can be treated as RLZ-encodings such that each factor is a single letter or an extension of an earlier factor by a single letter.

The basic idea behind lossless 1-dimensional compression is to make use of repetitions (e.g. in LZ-encoding), In abstract terms it can be represented as a sequence of recurrences: next portion of a text uses references to previously specified portions. The abstract formulation of such sequences of recurrences can be done using terminology of *straight-line* programs (SLP's, in short) , which are, informally speaking, sequences of equations involving variables and constants. Sequential algorithms for SLP-compressed matching were considered in [19]. We shall prove that RLZ-encodings can be easily transformed into SLP-encoding with logarithmic increase in size. The advantage of SLP-encodings is that it is much easier to design algorithms for such type of description, they play similar role as Chomsky normal forms for context-free grammars. SLP's are especially convenient tool to construct NC algorithm for counting number of occurrences of a pattern in a RLZ-compressed text using SLP's.

Another variation of Lempel-Ziv approach is the Lempel-Ziv-Welch compression (LZW, in short), which is one of the mostly used text compression and is frequently used under Unix. We refer to [23] for detailed definition. Basically it is a Restricted LZ compression when we can extend factors by single letters only. An efficient sequential algorithm for LZW-compressed matching was presented in [2].

The last compression method discussed in the paper is high compression of 2-dimensional texts in terms of simple 2-dimensional grammars, or straight-line programs (sequences of recurrences), see [5]. The 2-dimensional SLP's are equivalent to deterministic acyclic finite automata describing images. We change terminology of *states* to *variables*, or nonterminals. Using such terminology it is easier to design our algorithms and it unifies the treatment of one- and two-dimensional case. Finite automata are a well established tool for the compression of 2-dimensional texts (images), see [17, 12]. They can describe quite complicated images, for example deterministic automata can describe the Hilbert's curve with a given resolution. while finite weighted automata can describe even much more complicated curves, see also [9, 10].

Our **main results** are:

- P-completeness of LZ-compressed matching
- Almost optimal parallel algorithm for Restricted LZ-compressed matching and matching in texts described by straight-line programs
- Almost optimal parallel algorithm for LZW-compressed matching
- Efficient NC-algorithm counting number of occurrences of the pattern in RLZ-compressed texts
- Efficient parallel algorithm for 2-dimensional compressed matching in terms of recurrences or finite automata (in the sense of [16]).

2 P-completeness of LZ-compressed matching

The difference between sequential and NC-computations can be well demonstrated by the following problem: compute the symbol $T[i]$ where T is the text given in its LZ-encoding or RLZ-encoding, and i is given in binary. This task has a trivial sequential linear time algorithm but if we ask for an NC-algorithm for the same problem the situation is different, and it becomes quite difficult (P-complete for LZ-encoding and *possibly* P-complete for RLZ-encodings). Any straightforward attempt of using the doubling technique and squaring the matrix of positions fails, since the number of positions in the text whose compressed version has size n can be $\Omega(2^n)$. We use the following auxiliary problem called the *iterated mod problem*

Input: given positive integers $x,\ m_1, m_2, \ldots, m_k$
Output: *"yes"* if $((\ldots(x \ mod\ m_k)\ mod\ m_{k-1})\ldots\ mod\ m_1) = 0$.

The following result was proved in [18]

Lemma 2.1 *The iterated mod problem is P-complete.*

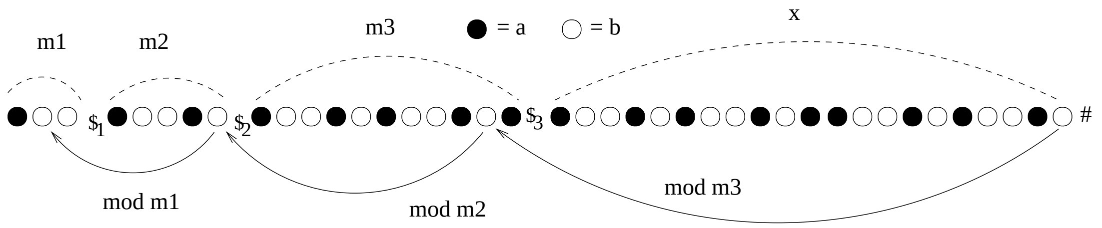

Figure 1: The structure of $w(x, m_1, m_2, \ldots, m_k)$ for $(x, m_1, m_2, \ldots, m_k) = (21, 3, 5, 11)$.

Theorem 2.2

(1) *The problem of testing for any occurrence of an uncompressed pattern in a LZ-compressed text is P-complete.*

(2) *The problem of computing a symbol on a given position in a LZ-compressed text is P-complete.*

Proof: Consider an instance $x,\ m_1, m_2, \ldots, m_k$ of the iterated mod problem. We can assume w.l.o.g. that $m_1 < m_2 < \ldots < m_k < x$.

Construct a string $w = w(x,\ m_1, m_2, \ldots, m_k)$ whose positions are counted starting from 0. Let $\$_1, \$_2, \ldots \$_k, \#$ be special different symbols. The structure of w is

$$w(x,\ m_1, m_2, \ldots, m_k) = w_1 \$_1 w_2 \$_2 w_3 \$_3 \ldots w_k \$_k v\#,$$

where: $|w_i| = m_i$ for $1 \leq i \leq m$, $|v| = x$, w_i is a period of $w_{i+1}$for $i < k$,
w_k is a period of v and $w_1 = ab^{m_1-1}$.

Claim 1. The following conditions are equivalent:

1. $((\ldots(x\ mod\ m_k)\ mod\ m_{k-1})\ldots\ mod\ m_1) = 0$
2. w contains the symbol a at the penultimate position of $|w|$
3. the pattern $a\#$ occurs in w

It is now sufficient to show the following fact.

Claim 2.. LZ-encoding of $w(x,\ m_1, m_2, \ldots, m_k)$ has number of bits polynomial with respect to the total number of bits in an instance $(x,\ m_1, m_2, \ldots, m_k)$, and it can be computed by an NC-algorithm.

Each w_{i+1} is of a form $w_i^t w_i'$, where t is a nonnegative integer and w_i' is a prefix (possibly full) of w_i. Let 2^j be the largest power of two not exceeding t. Assume the position of the first symbol of w_{i+1} is $r+1$, then encoding of the segment w_{i+1} is $[r - |w_i|, r-1]$, $[r+1, r+m_i]$, $[r+1, r+2m_i], \ldots [r+1, r+2^{j-1}m_i], [r+1, r+m_{i+1} - 2^j m_i]$.

For example the encoding of the whole word $w = w(69, 2, 3) = ab\$_1 aba\$_2 (aba)^{23} ab\#$ is

$a\ b\ \$_1\ [0,1]\ a\ \$_2\ [3,5]\ [7,9]\ [7,12]\ [7,18]\ [7,30],\ [7,29]\ \#.$

Such encodings can be easily constructed in parallel. This completes the proof. □

We show in the next section that the problem of RLZ-compressed matching is much easier and it is in NC, however we want to emphasize that in general NC-computations related to

RLZ-encodings are difficult. We conjecture that the following problems related to Restricted LZ-encodings are P-complete (the first two are P-complete for LZ-encodings).

1. Computing the i-th symbol of T when T is given by its RLZ-encoding.
2. The *fully compressed pattern-matching*: both text T and the pattern P are given in RLZ-compressed form.
3. Given two RLZ-encodings, do they describe the same text ?

Several P-complete problems related to text compression were presented in [11].

3 Naive NC-algorithm for RLZ-compressed matching

Despite the fact that computing a symbol on a given arbitrary chosen position of RLZ-encoded text is quite possibly P-complete we use NC-computation of symbols at some *easy* positions (close to defined later critical positions). The LZ-encoding partitions the texts into factors, we call here the beginning and starting positions of these factors *critical points*. We call a positions which is at distance at most m from a critical point a *working position*. Due to Lemma 3.1 it is enough to compute in NC all the symbols at *working positions*. This problem in P-complete for LZ-encoding but it is in NC for RLZ-encoding. We can use Observation 2.1 from [7], which basically says the following.

Lemma 3.1 *Assume that T is compressed using LZ or RLZ-encoding. Then the pattern P occurs in T iff it occurs within a distance m from a critical position.*
Assume w is a text compressed by a RLZ-encoding. For a position i in w denote by $LINK(i)$ the position which is computed as follows: if the factor contains the position i is a single symbol then $LINK(i) = i$, otherwise the factor containing i is a composition of a sequence of earlier factors. The position i corresponds to some position j in one of these factors, we set $LINK(i) = j$. We need to know lengths of factors to decide how they are split and in which component factor $LINK(i)$ is located.

Observe that if i is a working position then $LINK(i)$ is also in case of RLZ-encodings. It is the main property of RLZ from the point of view of NC algorithmics. Such property does not hold for LZ-encodings.

Algorithm *Naive RLZ-matching*;
Comment: given pattern P and a RLZ-encoding of T
(1) compute in parallel lengths of all factors;
Comment: it needs $O(n^3)$ processors
(2) **for each** working position i **do in parallel**;
compute $LINK(i)$
(3) compute the closure $LINK^*$ of the graph defined by LINK table;
(4) for each working position i compute the i-th symbol using $LINK^*(i)$
(5) **for each** critical position i **do in parallel**;
check if there is an occurrence of P overlapping i;
Comment: such occurrence contains only working positions;
(6) **return** $''Yes''$ if there is at least one occurrence;

Theorem 3.2 *The naive algorithm solves RLZ-compressed matching problem in a polylogarithmic time with $O(n^3 \cdot m^3)$ processors.*
Proof: There are the following tasks in the algorithm which need many processors:

1. Computation of lengths of factors, we can use parallel computation of straight-line programs containing only the + operation. This needs n^3 processors
2. Computation of $LINK^*$ needs a transitive closure of a directed graph with nm vertices, since we have $O(nm)$ working positions. A straightforward algorithm needs cubic number of processors.
3. Searching on each overlap individually needs $O(nm)$ processors, m processors for each of n critical positions. □

The number of processors can be easily reduced to $n^3 + nm$ but further reduction is more complicated.

Observation 3. It is not clear how to extend the naive algorithm to compute (by an NC algorithm) the number of all occurrences. The main difficulty is that now we have also to consider occurrences of the pattern in T which do not contain any *working position*. We shall see that straight-line programs approach is a convenient tool to solve it.

4 Almost optimal parallel RLZ-compressed matching

In this section we show that RLZ-compressed matching can be solved by a NC algorithm whose work is linear within a polylogarithmic factor.
Theorem 4.1 *The RLZ-compressed matching problem can be solved in $O(\log^2(m)+\log^2(n))$ time with $O(n)$ processors.*
LZW-encodings can be treated as special RLZ-encodings.
Corollary 4.2 *There is an almost optimal NC-algorithm for LZW-compressed matching.*

Before going into the proof of Theorem 4.1 we reduce RLZ-matching to searching patterns in texts with short recursive descriptions described by straight-line programs. A short description of one-dimensional texts is in a form of a sequence of recurrence equations, or (more formally) straight-line programs, see [19]. A *straight-line program* $\mathcal{R}$ is a sequence of assignment statements: $X_1 = expr_1;\ X_2 = expr_2; \ldots;\ X_n = expr_n$,
where X_i are variables and $expr_i$ is a symbol in Σ, or $expr_i = X_j \cdot X_k$, for some $j, k < i$.

We assume here that all variables are relevant, this means that we cannot remove any equation from the straight line program without affecting its value. It means, in terms of grammars, that there are no *useless* variables. Such short descriptions are equivalent to acyclic context-free grammars in Chomsky normal form generating a single text. Testing useless variables is P-complete for general context-free grammars but it can be shown that it is in NC if each variable appears only once on the left-hand side of a production, as it is in our case. Denote by R the string which is the value of the last variable X_n after the execution of the program $\mathcal{R}$. We identify variables with their values in the sequel.
Lemma 4.3 *RLZ-encodings can be transformed into SLP-encoding with logarithmic increase in size in $O(\log n)$ time with $O(n)$ processors.*
We assume now (to the end of this section) that we deal with SLP-encodings.

We say that a pattern P *occurs on an overlap* iff P occurs in a variable A in such a way that it starts in B and ends in C, where $A = BC$ is a rule of the straight-line program. The one-dimensional texts with short description have the following useful property

Lemma 4.4 *[overlap-lemma] For $|P| \geq 2$ each match of P in T occurs on an overlap.*

We say that a variable A is *small* iff the text generated by it does not exceed m.

The *tree-size* of a node of a directed acyclic graph G is the number of paths from this node to a leaf. If a graph corresponds to a circuit computing an expression then the tree-contraction approach can be used in a straightforward way to compute nodes whose tree-size is polynomial.

Lemma 4.5 *The exact lengths of all small variables in the straight line program $\mathcal{R}$ can be computed in $O(\log(m) + \log^2(n))$ time with $O(n)$ processors. The variables whose lengths are not computed are not small.*

Proof: We can replace concatenation of variables (whose values are strings) by addition of numbers (lengths of variables). We can view $\mathcal{R}$ as a directed acyclic graph G, in which variables correspond to nodes, the only operation in the nodes is addition and each variables has edges leading to the variables which are on the right-hand side of a corresponding equation.

Then we can use the tree contraction algorithm called *parallel pebble game*, see [14]. This tree contraction works as well for binary directed acyclic graph G (where the notion of the sons is well defined), and after $O(\log(m)$ parallel time the values of all variables are computed whose *tree-sizes* do not exceed m. In our case it is easy to see the following fact.

Claim. The tree-size of a node v corresponding to a variable x equals the length of the string generated from x.

Hence after logarithmic number of applications of our tree-contraction applied to directed acyclic graphs the lengths of all *small* variables are computed. The lengths of not computed variables are larger than m. This completes the proof. □

Proof: (of **Theorem 4.1**)

We apply first Lemma 4.5 to compute exact lengths of all *small* variables, if the length is not computed then it means that the length exceeds m. For each variable X denote by $LEFT(X)$ ($RIGHT(X)$) the largest suffix (prefix) of P which is a prefix (suffix) of X.

Claim 1. We can preprocess the pattern P in $O(\log m)$ time with linear number of processors in such a way that we can answer in logarithmic time each query of the form:
given i, j, p, q find r, s such that $P[i..j] \cdot P[p..q] = P[r..s]$, if such r, s exist.

Proof (of the claim).

We build the suffix tree of P in parallel using the algorithm from [4]. Each subword of P whose length is a power of two gets a name (the identical subwords have the same names). Using the shortcuts (whose lengths are powers of two) in the suffix tree we search for $P[i..j] \cdot P[p..q]$ in the suffix tree in logarithmic time [13]. We omit the details.

Claim 2.

We can compute $LEFT(A)$ and $RIGHT(A)$ for each variable A in $O(\log^2(m) \cdot \log(n))$ time with $O(n + m)$ processors.

Proof (of the claim). We only show how to compute $RIGHT(A)$ for each A. Denote by $RIGHT'(A)$ the subword of P which equals the value of A, or, if there is no such subword then $RIGHT'(A) = RIGHT(A)$. Denote also by $\otimes$ the operation of composing two subwords u, w of P. If uw is a subword of P then the result of $u \otimes w$ is this subword (given by the first and the last position), otherwise it is the largest suffix of uw which is a prefix of P. Consider the dependency graph G of a given straight line program describing T. G is a directed acyclic graph, each node (variable) has a left and right son, unless it is a leaf. For each variable X

let Y be the first variable when traversing from X using right branches until finding a node (variable) whose right son is a small variable. Let sons of Y by U, W, then we create an equation $X = U \otimes W$. The computation of Y for each X is a again reducible to some tree computations. In this way we get a new straight-line program $\mathcal{R}'$ which computes $RIGHT'(X)$ for each variable X. The graph of $\mathcal{R}'$ has the property that the tree-size of each right son is $O(m)$ since it corresponds to a small variable, consequently the following basic property of $\mathcal{R}'$ is: the tree-size of each variable with respect to $\mathcal{R}'$ is $O(n \cdot m)$.

Now we can compute $RIGHT'(X)$ for each variable X applying a tree-contraction to the graph of $\mathcal{R}'$. This graph is not a tree, but its tree-size is polynomial, so the number of iterations in the tree-contraction method applied to this graph is logarithmic. The operations $\otimes$ can be performed in $O(\log m)$ time, due to Claim 1 if it involves two subwords. If we have a prefix concatenated with a subword we can use border trees from [15]. The value of $RIGHT(X)$ can be recovered from $RIGHT'(X)$. This completes the proof of the claim.

For each variable A with the equation $A = B \cdot C$ we can check if occurrence of P overlaps splitting point of A by testing if P occurs in $RIGHT(A) \cdot LEFT(B)$. This can be done fast for all A's after preprocessing the pattern (using border trees of [15]). □

Theorem 4.6 *The number of occurrences and the first occurrence of the pattern in RLZ-encoded text can be found in polylogarithmic time with $O(n^3 + m)$ processors.*

Proof: First we show how to compute number of occurrences. At the beginning we compute the number $\#overlap\_occ\_in(X)$ of occurrences overlapping the splitting point of each variable X. Then the problem is reduced to the computation of the number of occurrences $\#X$ of each variable X in the generation tree of the text due to the following fact.

Claim. The number of occurrences of P equals $\sum_{X \text{ is a variable}} (\#X \cdot \#overlap\_occ\_in(X))$.

Assume all variables are relevant. For each variable X let $father(X)$ be the set of possible father variable of X in the generation tree. The numbers $\#X$ can be computed using NC-computation of straight-line programs involving the operation $+$ and using for each non-root variable the equation.

$$\#X \;=\; \sum_{Y \in fathers(X)} \#Y \qquad .$$

In the generation-tree for the terminal text we can search for the the first variable from the left and from the bottom such that the pattern occurs on an overlap in this variable. We call such variable a *good variable*. We have already shown that testing for an existence of an occurrence of a pattern on an overlap in each variable can be done within required complexities. Hence the problem is reduced to finding the good variable. First for each two variables we compute in parallel if it first occurs in the tree generated by the latter one. This can be done using a transitive closure of a graph G from the proof of Lemma 4.5. Then we compute sizes of all variables (not only small ones). The computation can be done by using a parallel transitive closure algorithm working in $\log^2 n$ time with $O(n^3)$ processors. Once we know the sizes of each variable and we know which of them contains the pattern on an overlap we can *travel* in the generation tree (or rather in a directed acyclic graph describing it succinctly) from the root down to the first good variable. Technically it can be done by building another graph G'. For each equation $A = BC$ we create edges $A \to B$ of weight 0 and the edge $A \to C$ of weight $|B|$. The computation of a first position of an occurrence is reduced to computing a shortest path from the root variable to a variable containing the pattern on an overlap. This completes the proof. □

5 Searching for two-dimensional patterns

We extend now the pattern-matching problem for shortly described texts to a two-dimensional case using description in terms of *restricted* grammars, straight-line programs and (equivalently) of deterministic acyclic automata, see [16]. Many interesting fractals drawn with a given finite resolution can be described using such formalism, e.g. Hilbert's curve, see [5]. The recurrences describing 2-dimensional texts in [5] are much stronger and for them the pattern-matching problem for highly compressible 2d-texts is NP-complete. Our description is more *restrictive*, since we always compose objects of a same shape. This implies existence of polynomial time algorithms, see [16]. We use the operation of constructing a square table of 4 smaller tables of a same shape. The operation *Compose* places the squares generated by X_j, X_k, X_s, X_r in a once fixed order. In this way the square of shape 2^t is composed of 4 already defined squares of shapes $2^{t-1} \times 2^{t-1}$. The length of a side of each square corresponding to a variable is of a form 2^t for some t, the number t is called the *rank* of this variable and also the rank of the 2-dimensional text generated by the it. A *2-dimensional simple straight-line program* $\mathcal{R}$ is a sequence of assignment statements where each variable has its *rank*: $X_1 = expr_1;\ X_2 = expr_2; \ldots;\ X_n = expr_n$,
X_i are variables and $expr_i$ is a symbol in Σ, in this case $rank(x_i) = 0$, or
$expr_i = Compose(X_j, X_k, X_s, X_r)$, for some $j, k, s, r < i$, and X_j, X_k, X_s, X_r are variables of the rank $t-1$ where $t = rank(x_i)$.

We assume here also that all variables are relevant, this means that we cannot remove any equation from the straight line program without affecting the final value.

Let T be a 2-dimensional text shortly described. We say that a pattern P has a *regular overlap property* for T iff it occurs inside the 2d-text generated by some variable X in such a way that it overlaps all squares generated by X_1, X_2, X_3, X_4, where $X = Compose(X_1, X_2, X_3, X_4)$.

Observation. It is possible that P occurs in T without regular overlap property.

The equivalent short description of 2d-text is possible in terms of deterministic acyclic automata. The states of the automaton correspond directly to the indices of variables. Consider the alphabet $\Sigma = \{0,\ 1,\ 2,\ 3\}$, the elements of which correspond to four quadrants of a square array. A word w of length k over Σ can be interpreted, in a natural way, as a unique address of a pixel x of a $2^k \times 2^k$ image (array), we write $address(x) = w$. The length k is called the *resolution* of the image. For a language $L \subseteq \Sigma^+$ denote by $image(L)$ the $2^k \times 2^k$ black-and-white image such that the color of a given pixel x is black iff $address(x) \in L$. Then our description of the language can be in terms of finite automata A. We define $image(A) = image(L(A))$, where $L(A)$ is the language accepted by A. A *regular block* of a $2^k \times 2^k$ 2d-text T is defined as follows. T is a regular block, and if B is a regular block then all its quadrants are also regular blocks. A regular block of the shape $2^t \times 2^t$ is said to be of *rank* t.

We can interpret each variable A in a straight-line program describing T as a name of a regular block of T: it is the 2d-text generated by A.
For each variable A by *small corners* of A (and of the regular block corresponding to A) we mean one of 4 corner sub-squares of area m.

Lemma 5.1
(1) *Small corners of all variables can be computed in $O(\log m)$ time with $O(n \cdot m)$ processors.*
(2) *We can check for a regularly overlapping occurrence in $O(\log m)$ time with $O(n \cdot m)$ processors.*
Proof: Use tree-contraction techniques similarly as in the proof of Theorem 4.1. □

However not each pattern contained in T has a regularly overlapping occurrence and we identify two additional types of overlapping occurrences.
A *vertically (horizontally) overlapping* occurrences of a pattern are the ones which touch the middle point of a (vertical or horizontal) boundary between two adjacent regular blocks of a same rank.

Lemma 5.2
(1) *A pattern P occurs in T iff there is a vertically, horizontally or regularly overlapping occurrence of P in T.*
(2) *If we know all pairs of names of (vertically or horizontally) adjacent regular blocks of a same rank then we can test occurrence of the pattern in $O(\log n + \log m)$ time with $O(n \cdot m)$ processors.*

Proof: We omit the proof in this version. □

Theorem 5.3
The pattern-matching problem for 2d-texts with short descriptions can be solved in
(1) $O(\log^2(n+m))$ *time with* $O(n^2 \cdot m + n^6)$ *processors, or*
(2) $O(n + \log m)$ *time with* $O(n \cdot (n+m))$ *processors.*
Proof: The main point is the computation of names of all adjacent blocks.

Claim. We can compute all pairs (A,B) such that regular blocks with names A, B are adjacent in T in $log^2 n$ time with $O(n^6)$ or in $O(n)$ time with $O(n^2)$ processors.
Proof: (of the claim) We show only the case of vertical overlapping, the horizontal case can be done similarly. Construct a 2-dimensional dependency graph $D2$, whose nodes are pairs of names of variables of a same rank. The edges correspond to the way of decomposing (according to the straight-line program describing T) adjacent blocks into sub-blocks of one smaller rank. We omit the formal description in this version. In the graph $D2$ a pair (A,B) is reachable from a node corresponding to a single variable iff the pair of regular block with names (A,B) occurs in T. The reachable pairs can be identified by computing in parallel the transitive closure of the dependency graph. We can do it fast in $log^2 n$ time with $O(n^6)$ processors or (using a parallel breadth-first search) in $O(n)$ time with $O(n^2)$ processors. □

Due to Lemma 5.1 we can compute efficiently all small corners of regular blocks. Afterwards we can detect all situations of regular overlaps and all situations of overlaps. Our version of the pattern-matching is reduced to $O(n^2)$ instances (since we have such number of pairs of names of blocks) of the classical two-dimensional pattern-matching for uncompressed patterns and uncompressed texts. We can use one of several NC algorithms for this problem. □

Acknowledgment.
We thank Paolo Ferragina for the help in the proof of Claim 1 inside the proof of Theorem 4.1.

References

[1] A. Amir and G. Benson, Efficient two dimensional compressed matching, *Proc. of 2nd IEEE Data Compression Conference*, pp. 279–288, March 1992.

[2] A. Amir, G. Benson, and M. Farach, Let sleeping files lie: Pattern matching in Z-compressed files, *Proc. of 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1994.

[3] A. Amir, G. Benson, and M. Farach, Optimal two-dimensional compressed matching, *Proc. of 21st International Colloquium on Automata, Languages, and Programming*, July 1994.

[4] A. Apostolico, C. Iliopoulos, G.M. Landau, B. Schieber, U. Vishkin, Parallel construction of a suffix tree with applications, Algorithmica 3 (1988) 347- 365

[5] P. Berman, M. Karpiński, L. Larmore, W. Plandowski, and W. Rytter, The complexity of pattern matching of highly compressed two-dimensional texts, Proc. of *8th Annual Symposium on Combinatorial Pattern Matching*, CPM'97, July 1997.

[6] M. Crochemore, W. Rytter, Text algorithms, Oxford University Press, 1994.

[7] M. Farach and M. Thorup, String Matching in Lempel-Ziv Compressed Strings, *Proc. 27th ACM Symposium on Theory of Computing*, pp. 703–713, 1994.

[8] L. Gąsieniec, M. Karpiński, W. Plandowski, and W. Rytter, Efficient algorithms for Lempel-Ziv encoding, *Proc. 5th Scandinavian Workshop on Algorithms Theory*, pp. 392–403, LNCS 1097, July 1996.

[9] K. Culik and J. Karhumaki, Finite automata computing real functions, *SIAM J. Comp* (1994).

[10] K. Culik and J. Kari, Image compression using weighted finite automata, *Computer and Graphics* 17, 305-313 (1993).

[11] S. De Agostino, P-complete problems in data compression, TCS 127, 181-186, 1994

[12] K. Culik and J. Kari, *Fractal image compression: theory and applications*, (ed. Y. Fisher), Springer Verlag 243-258 (1995).

[13] P. Ferragina, Personal communication

[14] A. Gibbons, W. Rytter. Efficient parallel algorithms, Cambridge Univ. Press 1988

[15] M. Gu, M. Farach, R. Beigel, An efficient algorithm for dynamic text indexing, SODA'94, 697-704

[16] J.Karhumaki, W.Plandowski, W. Rytter, Pattern matching for images generated by finite automata, FCT'97, in LNCS Springer Verlag 1997

[17] J. Kari, P. Franti, Arithmetic coding of weighted finite automata, *RAIRO Theor. Inform. Appl.* 28 343-360 (1994).

[18] H.J. Karloff, W.L. Ruzzo, The iterated mod problem, Inf. & Comp. 80(3) 193-204, 1989

[19] M. Karpiński, W. Rytter, and A. Shinohara, A pattern matching for strings with short description, *Proc. 6th Combinatorial Pattern Matching*, June 1995.

[20] A. Lempel and J. Ziv On the complexity of finite sequences, *IEEE Transactions on Information Theory*, pp. 22:75–81, 1976.

[21] M. Miyazaki, A. Shinohara, and M. Takeda, An Improved Pattern Matching for Strings in Terms of Straight-Line Programs, *8th Combinatorial Pattern Matching*, July 1997, LNCS 1264, pp. 1–11.

[22] W. Plandowski and W. Rytter, Application of Lempel-Ziv encodings to the solution of words equations, *Proc. of 25th International Colloquium on Automata, Languages, and Programming*, to appear.

[23] T.A. Welch, A technique for high performance data compression, *IEEE transactions on Computers*, pp. 17:8–19, 1984.

[24] J. Ziv and A. Lempel, A universal algorithm for sequential data compression, *IEEE Transactions on Information Theory*, pp. IT-23(3):337–343, 1977.