STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLIX**, Number 2, 2004

MAXIMAL PROCESSOR UTILIZATION IN PARALLEL QUADTREE-BASED FRACTAL IMAGE COMPRESSION ON MIMD ARCHITECTURES

BODÓ ZALÁN-PÉTER

Abstract. Since fractal image compression is computationally very expensive, some researchers tried to parallelize the encoding algorithm. Because this algorithm is applied independently for some image blocks, fractal image compression implicitly encompasses parallelism. The quadtree-based compression proceeds recursively and terminates when the previously fixed threshold remains *unexceeded*, therefore one cannot be able to calculate and store the whole domain pool for classification. This, however, can result some idle processors during the encoding, which is undesirable. In this paper a parallel implementation devoid of classification will be presented, keeping the number of idle processors at minimal.

1. Introduction

Belonging to the class of lossy data compressors, fractal image compression is based on self-similarity in real-world images, where "an image is modeled as the unique fixed point of a contractive operator on the space of images" [6]. Barnsley was who discovered and proved the most important theorem in 1985, namely the *Collage Theorem* [1, pp. 94–95], which serves as a basis for the fractal coding of images. Jacquin wrote down first the famous algorithm of fractal image compression, based on *partitioned iterated function systems* (PIFS). The algorithm partitions the image into smaller independent (range and domain) blocks, where for every range block the best matching domain block is needed to be found. The long search time makes the encoding problematic. Besides high compression ratios fractal image compression provides resolution independent image description, that is the reconstructed image can be zoomed-in without pixelisation.

Received by the editors: August 24, 2004.

2000 *Mathematics Subject Classification.* 65Y05, 94A08.

1998 *CR Categories and Descriptors.* D.1.3. [**Programming Techniques**]: Concurrent Programming – *Parallel programming*; I.4.2. [**Image Processing and Computer Vision**]: Compression – *Approximate methods*.

The main drawback of the proposed quadtree-based fractal image compressions on MIMD (Multiple Instruction stream and Single Data stream) architectures is that increasing the number of processors, after a while the gain from adding a new processor is almost zero. In this paper we present a simple algorithm for MIMD architectures trying to maximize processor utilization during the encoding phase.

The paper is organized in the following way: in Section 2 we present the procedure of fractal image compression with quadtrees. Details about the complexity of fractal encoding of images can be found in Section 3. In Section 4 we present some parallel implementations of fractal coding systems on MIMD architectures. In Section 5 the proposed parallel quadtree-based fractal image compression algorithm is presented in details. The test results and the conclusions can be found in Section 6.

2. Fractal Image Compression with Quadtrees

In fractal image compression the image is modeled as the unique fixed point of the contractive operator

$$W(\cdot) = \bigcup_{n=1}^{N} w_n(\cdot),$$

where w_n, $n = 1, 2, \ldots, N$ are contraction mappings, whose set is called a partitioned iterated function system (PIFS). The well known *copying machine* is an informal denomination of the mathematical structure called iterated function system (IFS). The single difference between IFS and PIFS is that the domains of the member functions of a PIFS are subsets of the plane on which the (P)IFS is defined. These structures simplify the fractal coding of *not really* self-similar sets. The domains of the transformations are called domain blocks (D_i), while the ranges are called range blocks (R_i), and we can write it in the following way:

$$w_i : D_i \to R_i, \quad i = 1, 2, \ldots, N.$$

Let T be an arbitrary grayscale image, that is $T : I^2 \to I$, where $I^2 = \{(x, y) \mid x, y \in [0, 1]\}$ and $I = \{x \mid x \in [0, 1]\}$. However, this space can be extended to arbitrary size, only in theory we work with these domains and ranges for the sake of simplicity. Then we seek such a contractive operator (a set of affine transformations) that

$$T = W(T) = \bigcup_{n=1}^{N} w_i(D_i).$$

To measure the distance between two blocks we need to find a feasible metric. In theory the supremum metric is used, which is easy to work with, but not so advantageous in practice, because it takes only one point from the image, consequently is not relevant for the whole image or image block. In practice the RMS (Root-Mean-Square) metric is used,

$$d_{RMS}(T,T') = \sqrt{\frac{1}{n}\sum_{(x,y)\in I^2}(T(x,y)-T'(x,y))^2}, \quad \forall T,T' \in \tau,$$

where $\tau = \{T : I^2 \to I\}$ denotes the space of digital images. The metric d_{sup} is equivalent with metric d_{RMS}.

To guarantee the z-contractivity of the $w_1,\dots,w_n$ three-dimensional transformations [2, pp. 12–13], in case of grayscale images we can use transformations of the form

$$w_i\begin{pmatrix} x\\ y\\ z\end{pmatrix} = \begin{pmatrix} a_i & b_i & 0\\ c_i & d_i & 0\\ 0 & 0 & s_i\end{pmatrix}\begin{pmatrix} x\\ y\\ z\end{pmatrix} + \begin{pmatrix} e_i\\ f_i\\ o_i\end{pmatrix},$$

where s_i controls the contrast and o_i controls the brightness of the transformation [4, pp. 11, 51]. The encoding algorithm consists of finding for all the range blocks (resulting from the used partition scheme – rectangular, quadtree, horizontal-vertical, Delaunay-triangular, etc.) such a domain block that the distance between them be minimal or at least smaller than a predetermined threshold. Usually the size of the domain blocks is chosen to be greater than that of the range blocks; comparison is realized by subsampling.

If $a_1, a_2, \dots, a_n$ denote the pixel intensities from the set D_i and $b_1, b_2, \dots, b_n$ from the set R_i ($|D_i| = |R_i| = n$, i.e. their cardinal numbers are equal), then we search for s, o so that the following expression be minimal:

$$R = \sum_{i=1}^{n}\left[(s\cdot a_i + o) - b_i\right]^2.$$

The optimal values of the contrast scaling (s) and brightness (luminance) shift (o) are calculated from the partial derivatives of the above expression. That is,

$$o = \frac{1}{n}\left(\sum_{i=1}^{n} b_i - s\sum_{i=1}^{n} a_i\right), \quad \text{and} \quad s = \frac{n\sum_{i=1}^{n} a_i b_i - \sum_{i=1}^{n} a_i \sum_{i=1}^{n} b_i}{n\sum_{i=1}^{n} a_i^2 - \left(\sum_{i=1}^{n} a_i\right)^2}.$$

Substituting these expressions into the initial formula we get

$$R = \frac{1}{n}\left[\sum_{i=1}^{n} b_i^2 + s\left(s\sum_{i=1}^{n} a_i^2 - 2\sum_{i=1}^{n} a_i b_i + 2o\sum_{i=1}^{n} a_i\right) + o\left(no - 2\sum_{i=1}^{n} b_i\right)\right].$$

The distance between two blocks is given by $\sqrt{R}$.

The fractal image encoding algorithm can be summarized as follows:

```
Let us determine a partitition for image $T$, made up of
range blocks $R_i$, such that $T = \bigcup R_i$
Fix a $t$ tolerance level
For all $R_i$ do:
    Let us find the domain block $D_i$ for which $d(R_i, D_i)$ is
    minimal or $d(R_i, D_i) < t$
    Store the transformation $w_i$ and the coordinates of $D_i$
End For
```

The quadtree partition is the most popular scheme in the fractal coding literature. It offers an adaptive partition, which gives better approximation than the fixed size. The partition process is not separated from the encoding step. Its name comes from the modeling (Fig. 1). The root of the tree is the whole, *unpartitioned* image. At the first level the image is divided into four equal parts. For all the ranges we scan the image (domain pool) for a domain block (usually twice the range size), which is very close, similar to the current range block. If the distance between the range and the transformed domain block is below a preselected treshold, than we store the domain coordinates and the transformation on the pixel values. If not, we divide the current range block into four equal quadrants, which means adding four childnodes to this range block in the tree representation. Then for each of the ranges the previous process is repeated.

The tree we obtain is called a quadtree, because each node of the tree can have

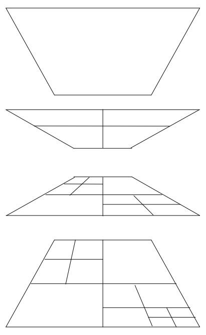

FIGURE 1. Quadtree decomposition.

either four subnodes or not any.

3. About the Complexity of Fractal Coding

Consider an $n \times n$ pixel image being encoded with fixed size partition and full-search (full-search means that no tolerance level is used). Let the size of a range block be $n_r \times n_r$ and the size of a domain block $n_d \times n_d$ (the size of the domain block is often chosen to be twice the range size, that is $n_d = 2 \cdot n_r$). It is easy to calculate that the number of ranges is $\lfloor n/n_r \rfloor^2$ and the number of domains $(n - n_d + 1)^2$. Then the complexity of the encoding procedure will be

$$\begin{gathered} nr\_of\_isometries \cdot \lfloor n/n_r \rfloor^2 \cdot (n - n_d + 1)^2 = \\ nr\_of\_isometries \cdot \mathcal{O}(n^2) \cdot \mathcal{O}(n^2) = \mathcal{O}(n^4) \end{gathered}$$

To illustrate how computationally expensive the encoding step is, let $n = 256$, $n_r = 8$, $n_d = 16$. Then, to encode this image $8 \cdot 1024 \cdot 58\,081 = 475\,799\,552$ comparison steps are needed and, as we have seen, one domain-range comparison is quite computationally expensive.

Ruhl and Hartenstein proved that finding an optimal fractal code is an NP-hard problem. They proved this by reducing the MAXCUT problem – which is NP-hard – to FRACCODE. The proof and the accurate definition of these decision problems can be found in [9]. In this paper they also proved that collage coding is not a ρ-approximating algorithm. That is, for every $\rho \in \mathbf{R}^+$ exists a signal (i.e. image) Γ and a transformation $g \in \pi$ (π is the set of possible fractal codes for the signal Γ) such that

$$\|\Omega_f - \Gamma\|^2 > \rho \cdot \|\Omega_g - \Gamma\|^2,$$

where f is the transformation that gives the best (minimal) collage error, $\|\Gamma - f(\Gamma)\|$; Ω_x is the attractor of the fractal code x.

4. Parallel MIMD Fractal Image Compression

There are two main algorithm classes for fractal image compression on MIMD architectures. The first class includes those algorithms which stores the whole image on each PE (Processing Element), namely when all the PEs have enough memory to have a local copy of the image. Thus each PE has the whole domain pool at its disposal. To each PE a subset of the range blocks is assigned, which can be made statically or dynamically. The second class includes those algorithms which distributes the domain pool among PEs, due either to memory lack or to other reasons. In [5] a more detailed class decomposition can be found.

Jackson and Tinney [7, 8] reported three generic schemes for parallel fractal image coding. In the following subsections these schemes will be presented.

4.1. **Static Load Allocation.** Static load allocation means that the assignment of the tasks, jobs is made at the beginning and no other modifications can be made afterwards. Due to the static property of this model, only fixed size partition schemes can be used, or those adaptive techniques, where the partition can be realized before, independently of the encoding process.

The most simpler solution is to distribute the range blocks evenly across the PEs. If we have n_p processors, then the work needed to be done by a PE is the $\frac{1}{n_p}$th of the work done in the sequential case. The speedup can be calculated easily here, that is the speedup will be n_p. But the experimental results show that rarely will the algorithm achieve this speedup, because for a range block a matching can be found quickly, while for another range block to find a good matching requires to scan almost the whole domain pool. If we had known something about the *complexity* of the range blocks (a range block is said to be complex if it is difficult to find a match for), then the ranges could have been distributed evenly across the PEs, according to their complexity. That is, a slave processor gets a small number of range blocks to be processed if they are complex, while another PE receives a larger number of simpler range blocks.

M. Chady [3] mentions other two factors which can cause the speedup to fall below the expected value. If we use a classification scheme, then the order of the classes is very important; the common classes should be placed before the uncommon ones, if not, the comparison process to classify a range block will be slowed down. Although if we talk about classification schemes, one cannot realize such a classification which provides equal comparison steps for each range block to classify, except if classification is done in parallel, but this requires as many idle (free) processors as many classes we have.
Chady mentions another problem which could slow down the encoding in case of quadtree partition scheme, but of course quadtree partition cannot be applied within static load allocation, because one can never predict how many range blocks will be finally and where they will be.

For the reasons mentioned above, this solution cannot guarantee uniform workload distribution.

4.2. **Dynamic Load Allocation.** Dynamic load allocation is often called *load balancing*. Load balancing means distributing the tasks evenly through the processors so that no processing element is overloaded. Load balancing technique is used especially when it is difficult to predict the number of tasks or the complexity of a task (time needed to perform the task).

The following scheme can be used as for fixed size partitions as for adaptive partition schemes like using quadtrees. Jackson defines this method for fixed size partitions in [8]. In dynamic load allocation there is a master or host processor which distributes the tasks among the other (slave) processors (Fig. 2). The master has two queues:

- queue of tasks (range blocks, waiting to be processed)
- queue of slaves (idle processors).

Using fixed size partitions the user can determine the so called package size, i.e. the size of individual assignments allocated by the master by specifying the number of the range blocks per allocation. The master makes the assignments, assigns a task to a slave until there are no more idle slave processors or range blocks to be processed. As a task is assigned to a slave, both the task and the slave is removed from their queues. On each return the master assigns a new task to the PE which returned the result.
Using quadtree partition scheme the differences are very small. The slave needs also to return a value besides the other parameters, which tells the master if a good matching was found or not. If there was a matching found and if the queue of tasks is empty, then the slave is placed back into the queue and the result is stored. If a matching was found and the queue of tasks is not empty, then a new task is assigned to the slave. If there was not found any suitable match, then the master divides the returned range block into four equal subsquares and places back both the range blocks and the slave into the corresponding queues and assigns a task to a slave until one of the queues becomes empty.

4.3. **Dynamic Allocation with Circulating Pipeline Processing.** In this configuration the slave processors communicate in a circulating pipeline fashion (Fig. 2). The domain pool is distributed among the slave processors. Like in the previous scheme the master maintains two queues. The assignments are transmitted from the master PE to idle slave PEs. When a task enters the pipeline enters

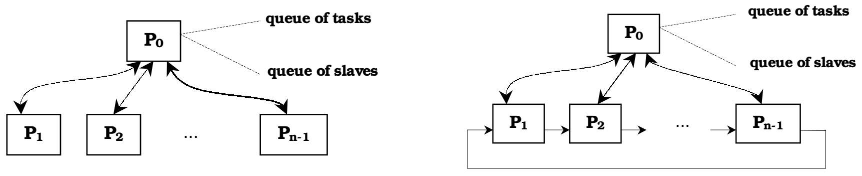

FIGURE 2. The structure of the PEs in dynamic load allocation and circulating pipeline schemes.

with a tag in which the number of the entering PE is stored. A range block will circulate in the pipeline until either a good match is found or all the pipeline nodes have been visited. The tag carrying the number of the entering node is used to check whether the task have visited all the PEs in the pipeline.

This configuration can be used both for fixed size and adaptive partition schemes.

5. Parallel Fractal Image Compression with Maximal Processor Utilization

Using classification schemes in quadtree-based fractal image compression algorithms on MIMD computers may degrade the performance of the encoding. According to Chady [3] the encoding proceeds in phases because the domain pool needs to be calculated for classification, the size of which grows exponentially as we proceed down in the quadtree. Therefore one cannot store the whole domain pool for a quadtree-partitioned image, but for example for one level only, which incidentally gives a good order for storing the parameters. However, there will be a certain period of time when many processors will be idle, since finding a good matching for a more complex range block may require much more computation, while the other processors have to wait for this PE until it terminates searching to proceed to the next quadtree level.
Avoiding classification the utilization of the processors can be increased. In this part we present this algorithm, the main idea being that the assignments are made immediately when new tasks arrive to the master PE. Besides this two recursive algorithms will be given for storing the partition efficiently.

The master uses a temporary file, where the results returned by the slave processes are stored. This is needed, because we want to save some space in the final, compressed file. The master sends the jobs to the slaves, but it is unknown which one will finish sooner, so if we want to save some space with storing no domain coordinates, first we need some space to store the temporary data (instead of using files one can use for example binary trees). When the master divides a range block, always assigns a unique string to the blocks, according to the position of it. For example the string `114` uniquely determines the range block with size $\frac{n}{4} \times \frac{n}{4}$ (if the image is of size $n \times n$) which is the lower-right quadrant of the upper-left quadrant of the upper-left quadrant of the image (see Fig. 3 and 4).

The scheme of the algorithms are presented hereinafter.

Master:

```
        Create the queues task and slave
        Read the image parameters (size)
```

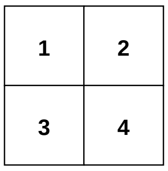

FIGURE 3. The numbering of the quadrants.

```
Put the four initial quadrants into *task*
Put the processor IDs into *slave*
nr_of_ranges := 4
While $task \neq \emptyset$ and $slave \neq \emptyset$ do:
    Get the first task and the first free slave
    Send the task to the slave
End While
While nr_of_ranges≠0 do:
    Block until receive the parameters from a slave
    Put the slave back to *slave*
    If a good matching was found, then:
         nr_of_ranges := nr_of_ranges-1
         Insert the returned parameters into the temporary
         file with unique string tag
    Else
         Divide the range block into four quadrants
         Put them into *task*
         nr_of_ranges := nr_of_ranges+4
    End If
    While $task \neq \emptyset$ and $slave \neq \emptyset$ do:
         Get the first task and the first free slave
         Send the task to the slave
    End While
    If nr_of_ranges = 0, then:
         Send to each process the terminate-message
         Sort the temporary file after the unique string
         tag (this can be made in parallel)
         Write the image/compression parameters to the
         final file
```

```
            Write the partition table to the file
            Copy the needed transformation parameters from the
            temporary file to the final one
        End If
    End While
```

Slaves:

```
    Read the image data
    While TRUE do:
        Block until receive some task
        If the terminate-message was received, then:
            break
        End If
        Search for a matching domain block
        Send to the master process the obtained parameters from
        the search
    End While
```

After the slave processors had been finished their work (they have got the terminate-message from the master), the results are stored in the temporary file or structure. A record contains the range coordinates, the domain coordinates and the parameters of the pixel intensity transformation (s, o). Besides these, all of the records (range blocks) have a unique string, after which the data can be sorted. Then we need a so called *binary partition table*, based on which we will write a recursive function which will draw the decompressed image in the decoder. In this case we don't have to store the coordinates of the range blocks, and thus we can save space.
Suppose that we have the following simple partition represented by strings

$$111 \quad 112 \quad 113 \quad 114 \quad 12 \quad 13 \quad 14 \quad 2 \quad 3 \quad 4.$$

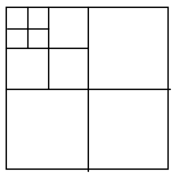

FIGURE 4. Visualization of the example.

Then the corresponding binary partition will be

$$(1(1(0000)000)000).$$

This can be constructed by the following recursive algorithm.

Partition writing algorithm:

```
procedure write_partition(length)
Begin
    For i=1, i<=4 do:
         if |buf.ustring| >length, then:
             write(1)
             write_partition(length+1)
         else write(0)
         if i<4, then read(buf)
    End For
End
```

```
call:  read(buf); write_partition(1)
```

In the decoder (viewer) a drawing function is needed, which visualizes the decoded picture calling the true drawing function.

Decompression algorithm based on the binary quadtree data:

```
procedure draw(x,y,rsize)
Begin
    read(buf)
    If buf=1 then draw(x,y,rsize/2)
    Else paint(x,y,rsize)
    read(buf)
    If buf=1 then draw(x+rsize,y,rsize/2)
    Else paint(x+rsize,y,rsize)
    read(buf)
    If buf=1 then draw(x,y+rsize,rsize/2)
    Else paint(x,y+rsize,rsize)
    read(buf)
    If buf=1 then draw(x+rsize,y+rsize,rsize/2)
    Else paint(x+rsize,y+rsize,rsize)
End
```

```
call:  draw(0,0,squaresize/2)
```

In the above algorithm the `read` function reads a bit from the binary partition into the variable `buf`. The function `paint` realizes the drawing of a range block. The variable `rsize` contains the vertical (horizontal) height (width) of the actual range block. The working of the function is simple: if we read 0 from the partition table we draw the corresponding range block, if the bit we have read was 1, then we call the function recursively with `rsize/2`. The range coordinates we call with depend on where the value 1 was read.

6. Test Results and Conclusions

The application was written in C/C++ under IRIX64 on the SGI Origin 3800 supercomputer (shared memory MIMD architecture with 128 R12000 400 MHz processors) situated at the Johannes Kepler Universität in Linz, Austria.

The tests were performed for two different pictures, which are not so relevant to show here, just to mention that the first one is a real-world image, while the second one is artificial. Although the results obtained are quite similar, we will present them separately. Moreover the test results were rather similar to those obtained using classification.

For measuring execution time and processor utilization we used the `timex` command under the IRIX64 operation system. This command can be parametrized to show the execution time for the "whole command" and among other things to show the execution time and the *hog factor* for each process(or). The hog factor gives the processor utilization – a real number between 0 and 1; it is calculated using the formula (total CPU time)/(elapsed time).

The plots (Fig. 5 and 6) were created using Mathematica. At the x-coordinate $x = 2$ we see the speedup (which is 1) and the processor utilization (which should be 1) of the sequential case, because there always have to be a master processor due to the used configuration.

In this paper we outlined the method of fractal image compression and the adaptive quadtree partition scheme. We discussed and analyzed the three main parallel distribution scheme applied for fractal encoding. The algorithm given in Section 5 avoids classification, but uses temporary data structure for efficient storage in the final, compressed file. We also gave an algorithm for the construction of the binary partition table. The results obtained show almost linear speedup up to a certain number of processors, depending on the complexity and size of the image

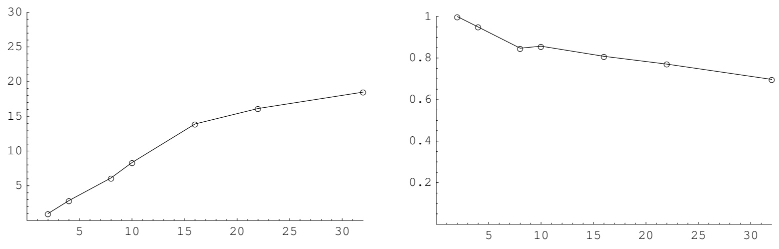

FIGURE 5. The speedup and average processor utilization results for the first image.

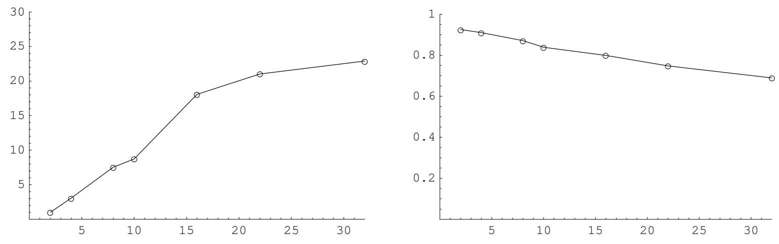

FIGURE 6. The speedup and average processor utilization results for the second image.

being encoded. Although the implemented algorithm performed quite efficient, a better parallelization would be needed to be worth using such an architecture.

REFERENCES

[1] M. F. Barnsley, *Fractals Everywhere*, second ed., Morgan Kaufmann, 1993.

[2] Z.-P. Bodó, *Parallel Fractal Image Compression*, Master Thesis, Babeş-Bolyai University, Faculty of Mathematics and Computer Science, Cluj-Napoca, 2004.

[3] M. Chady, *Application of the Bulk Synchronous Parallel Model in Fractal Image Compression*, School of Computer Science, University of Birmingham, http://citeseer.ist.psu.edu/255267.html.

[4] Y. Fisher (ed.), *Fractal Image Compression - Theory and Application*, Springer-Verlag, New York, 1996.

[5] J. Hämmerle, A. Uhl, *Parallel Algorithms for Fractal Image Coding on MIMD Architectures*, in Proceedings of the First International Conference on Visual Information Systems (Visual'96), Melbourne, February 1996, pp. 182–191.

[6] H. Hartenstein, M. Ruhl, D. Saupe, *Region-Based Fractal Image Compression*, IEEE Trans. on Image Process., Vol. 9, No. 7 (2000), pp. 1171–1184.

[7] D. J. Jackson, G. S. Tinney, *Fractal Image Compression Using a Circulating Pipeline Computation Model*, Technical Report UA-CARL-95-DJJ-01, Computer Architecture Research Laboratory, The University of Alabama, March 1995.

[8] D. J. Jackson, G. S. Tinney, *Performance Analysis of Distributed Implementations of a Fractal Image Compression Algorithm*, Concurrency: Practice and Experience, 8(5) (June 1996), pp. 357–380.

[9] M. Ruhl, H. Hartenstein, *Optimal Fractal Coding is NP-Hard*, in Proceedings DCC'97 Data Compression Conference, J. A. Storer, M. Cohn, eds., IEEE Computer Society Press, March 1997, pp. 261–270.

Babeş-Bolyai University, Faculty of Mathematics and Computer Science, Cluj-Napoca, Romania

E-mail address: `zpbodo@yahoo.com`