Compressing Bi-Level Images by Block Matching on a Tree Architecture

Sergio De Agostino

Computer Science Department
Sapienza University
Via Salaria 113, 00198 Roma, Italy
deagostino@di.uniroma1.it

Abstract. A work-optimal $O(\log M \log n)$ time parallel implementation of lossless image compression by block matching of bi-level images is shown on a full binary tree architecture under some realistic assumptions, where n is the size of the image and M is the maximum size of the match. Decompression on this architecture is also possible with the same parallel computational complexity. Such implementations have no scalability isuues.

Keywords: lossless compression, sliding dictionary, bi-level image, tree architecture

1 Introduction

Storer suggested that fast encoders are possible for two-dimensional lossless compression by showing a square greedy matching heuristic for bi-level images, which can be implemented by a simple hashing scheme [6]. Rectangle matching improves the compression performance, but it is slower since it requires $O(M \log M)$ time for a single match, where M is the size of the match [7]. Therefore, the sequential time to compress an image of size n by rectangle matching is $\Omega(n \log M)$.

The technique is a two-dimensional extension of LZ1 compression [5]. Simple and practical heuristics exist to implement LZ1 compression by means of hashing techniques [2], [9], [10]. The hashing technique used for the two-dimensional extension is even simpler.

Among the different ways of reading an image, we assume that the rectangle matching compression heuristic is scanning an $m \times m'$ image row by row (*raster* scan). A 64 K table with one position for each possible 4×4 subarray is the only data structure used. All-zero and all-one rectangles are handled differently. The encoding scheme is to precede each item with a flag field indicating whether there is a monochromatic rectangle, a match, or raw data. When there is a match, the 4×4 subarray in the current position is hashed to yield a pointer to a copy. This pointer is used for the current rectangle greedy match and then replaced in the hash table by a pointer to the current position. As mentioned above, the procedure for computing the largest rectangle match with left upper corners in positions (i, j) and (k, h) takes $O(M \log M)$ time, where M is the size of the match. This procedure can be used for computing the largest monochromatic rectangle in a given position (i, j) as well. If the 4×4 subarray in position (i, j) is monochromatic, then we compute the largest monochromatic rectangle in that position. Otherwise, we compute the largest rectangle match in the position provided by the hash table and update the table with the current position. If the subarray is not hashed to a pointer, then it is left uncompressed and added to the hash table with its current position. The positions covered

Sergio De Agostino: Compressing Bi-Level Images by Block Matching on a Tree Architecture, pp. 137–145.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2 © Czech Technical University in Prague, Czech Republic

by matches are skipped in the linear scan of the image and the sequential time to compress an image of size n by rectangle matching is $\Omega(n \log M)$. We want to point out that besides the proper matches we call a match every rectangle of the parsing of the image produced by the heuristic. We also call pointer the encoding of a match.

The analysis of the running time of these algorithms involve a so called *waste factor*, defined as the average number of matches covering the same pixel. In [7], it is conjectured that the waste factor is less than 2 on realistic image data. Therefore, the square greedy matching heuristic takes linear time while the rectangle greedy matching heuristic takes $O(n \log M)$ time. On the other hand, the decoding algorithms are both linear.

Parallel coding and decoding algorithms were shown in [3] requiring $O(\log M \log n)$ time and $O(n/\log n)$ processors on the PRAM EREW, mesh of trees, pyramidal, and multigrid architectures. The parallel encoder and decoder on the pyramid and the multigrid require some realistic assumptions. Under the same realistic assumptions, we show in this paper how to implement such encoder/decoder with the same parallel complexity on a full binary tree architecture. In section 2, we explain the block matching heuristic. In section 3, we describe scalable algoritms for coding and decoding bi-level images compressed by block matching on an exclusive read, exclusive write shared memory parallel machine. In section 4, we show how such parallel implementations can be run on a tree architecture. Conclusions and future work are given in section 5.

2 The Block Matching Heuristic

The compression heuristic scans an image row by row. We denote with $p_{i,j}$ the pixel in position (i, j). The procedure for finding the largest rectangle with left upper corner (i, j) that matches a rectangle with left upper corner (k, h) is described in figure 1.

```
w = k;
r = i;
width = m;
length = 0;
side1 = side2 = area = 0;
repeat
    Let p_{r,j} ··· p_{r,j+ℓ−1} be the longest match in (w, h) with ℓ ≤ width;
    length = length + 1;
    width = ℓ;
    r = r + 1;
    w = w + 1;
    if (length ∗ width > area) {
        area = length ∗ width;
        side1 = length;
        side2 = width;
    }
until area ≥ width ∗ (i − k + 1) or w = i + 1
```

Figure 1. Computing the largest rectangle match in (i, j) and (k, h).

At the first step, the procedure computes the longest possible width for a rectangle match in (i, j) with respect to the position (k, h). The rectangle $1 \times \ell$ computed at the first step is the current rectangle match and the sizes of its sides are stored in *side*1 and *side*2. In order to check whether there is a better match than the current one, the longest one-dimensional match on the next row and column j, not exceeding the current width, is computed with respect to the row next to the current copy and to column h. Its length is stored in the temporary variable *width* and the temporary variable *length* is increased by one. If the rectangle R whose sides have size *width* and *length* is greater than the current match, the current match is replaced by R. We iterate this operation on each row until the area of the current match is greater or equal to the area of the longest feasible *width*-wide rectangle, since no further improvement would be possible at that point. For example, in figure 2 we apply the procedure to find the largest rectangle match between position $(0, 0)$ and $(6, 6)$.

0	0	1	0	1	1	0	1	0	0	0	0	1	1	1	step 1
0	0	1	1	1	0	0	1	0	0	0	0	0	1	0	step 2
1	0	1	1	1	0	0	1	0	1	0	0	1	1	1	step 3
0	1	1	0	1	1	0	0	0	0	0	0	0	1	1	step 4
0	1	1	0	1	0	0	1	0	0	0	0	0	0	1	step 5
0	0	0	0	1	1	0	1	1	0	0	0	1	1	1	step 6
0	0	1	0	1	1	0	0	1	0	1	1	1	1	1	step 1
0	0	1	0	1	1	0	0	1	1	0	0	1	1	1	step 2
0	0	1	0	1	1	1	0	0	0	0	0	1	1	1	step 3
0	0	1	0	1	1	0	1	0	0	0	0	1	1	1	step 4
0	0	1	0	1	1	0	1	0	0	0	0	1	1	1	step 5
0	0	1	0	1	1	0	1	0	0	0	0	1	1	1	step 6
0	0	0	0	1	1	0	1	1	0	0	0	1	1	1	

Figure 2. The largest match in (0,0) and (6,6) is computed at step 5.

A one-dimensional match of width 6 is found at step 1. Then, at step 2 a better match is obtained which is 2×4. At step 3 and step 4 the current match is still 2×4 since the longest match on row 3 and 9 has width 2. At step 5, another match of width 2 provides a better rectangle match which is 5×2. At step 6, the procedure stops since the longest match has width 1 and the rectangle match can cover at most 7 rows. It follows that 5×2 is the greedy match since a rectangle of width 1 cannot have a larger area. Obviously, this procedure can be used for computing the largest monochromatic rectangle in a given position (i, j) as well.

As mentioned in the introduction, the procedure for computing the largest rectangle match takes $O(M \log M)$ time, where M is the size of the match. The positions covered by matches are skipped in the linear scan of the image and the sequential time to compress an image of size n by rectangle matching is $\Omega(n \log M)$. The analysis of the running time of this algorithm involve a so called *waste factor*, defined as the average number of matches covering the same pixel. In [7], it is conjectured that

the waste factor is less than 2 on realistic image data. Therefore, the square greedy matching heuristic takes linear time while the rectangle greedy matching heuristic takes $O(n \log M)$ time. On the other hand, the decoding algorithms are both linear.

3 A Massively Parallel Block Matching Algorithm

Coding and decoding algorithms are shown in [3] on the PRAM EREW, mesh of trees, pyramidal, and multigrid architectures, requiring $O(\log M \log n)$ time and $O(n/\log n)$ processors. The pyramid and multigrid implementations need some realistic assumptions. Under the same realistic assumptions, we show in the next section how to implement such encoder/decoder with the same parallel complexity on a full binary tree architecture. In this section, we present the PRAM EREW encoder/decoder. These algorithms can be implemented in $O(\alpha \log M)$ time with $O(n/\alpha)$ processors for any integer square value $\alpha \in \Omega(\log n)$.

To achieve sublinear time we partition an $m \times m'$ image I in $x \times y$ rectangular areas, where x and y are $\Theta(\alpha^{1/2})$. In parallel for each area, one processor applies the sequential parsing algorithm so that in $O(\alpha \log M)$ time each area will be parsed in rectangles, some of which are monochromatic. Before encoding we wish to compute larger monochromatic rectangles.

3.1 Computing the Monochromatic Rectangles

We compute larger monochromatic rectangles by merging adjacent monochromatic areas without considering those monochromatic rectangles properly contained in some area. Such limitation has no relevant effect on the compression ratio.

We denote with $A_{i,j}$ for $1 \leq i \leq \lceil m/x \rceil$ and $1 \leq j \leq \lceil m'/y \rceil$ the areas into which the image is partitioned. In parallel for $1 \leq i \leq \lceil m/x \rceil$, if i is odd, a processor merges areas $A_{2i-1,j}$ and $A_{2i,j}$ provided they are monochromatic and have the same color. The same is done horizontally for $A_{i,2j-1}$ and $A_{i,2j}$. At the k-th step, if areas $A_{(i-1)2^{k-1}+1,j}$, $A_{(i-1)2^{k-1}+2,j}, \ldots, A_{i2^{k-1},j}$, with i odd, were merged, then they will merge with areas $A_{i2^{k-1}+1,j}$, $A_{i2^{k-1}+2,j}, \ldots, A_{(i+1)2^{k-1},j}$, if they are monochromatic with the same color. The same is done horizontally for $A_{i,(j-1)2^{k-1}+1}$, $A_{i,(j-1)2^{k-1}+2}, \ldots, A_{i,j2^{k-1}}$, with j odd, and $A_{i,j2^{k-1}+1}$, $A_{i,j2^{k-1}+2}, \ldots, A_{i,(j+1)2^{k-1}}$. After $O(\log M)$ steps, the procedure is completed and each step takes $O(\alpha)$ time and $O(n/\alpha)$ processors since there is one processor for each area. Therefore, the image parsing phase is realized in $O(\alpha \log M)$ time with $O(n/\alpha)$ processors on an exclusive read, exclusive write shared memory machine.

3.2 The Parallel Encoder

We derive the sequence of pointers from the image parsing computed above. In $O(\alpha)$ time with $O(n/\alpha)$ processors we can identify every upper left corner of a match (proper, monochromatic, or raw) by assigning a different segment of length $\Theta(\alpha)$ on a row to each processor. Each processor detects the upper left corners on its segment. Then, by parallel prefix we obtain a sequence of pointers decodable by the decompressor paired with the sequential heuristic. However, the decoding of such sequence seems hard to parallelize. In order to design a parallel decoder, it is more suitable to produce the sequence of pointers by a raster scan of each of the areas into which the image was originally partitioned, where the areas are ordered by a raster

scan themselves. Again we can easily derive the sequence of pointers in $O(\alpha)$ time with $O(n/\alpha)$ processors by detecting in each area every upper left corner of a match and producing the sequence of pointers by parallel prefix.

As mentioned in the introduction, the encoding scheme for the pointers uses a flag field indicating whether there is a monochromatic rectangle (0 for the white ones and 10 for the black ones), a proper match (110), or raw data (111). For the feasibility of the parallel decoder, we want to indicate the end of the encoding of the sequence of matches with the upper left corner in a specific area. Therefore, we change the encoding scheme by associating the flag field 1110 to the raw match so that we can indicate with 1111 the end of the sequence of pointers corresponding to a given area. Moreover, since some areas could be entirely covered by a monochromatic match 1111 is followed by the index associated with the next area by the raster scan. The pointer of a monochromatic match has fields for the width and the length while the pointer of a proper match also has fields for the coordinates of the left upper corner of the copy in the window. In order to save bits, the value stored in any of these fields is the binary value of the field plus 1 (so, we employ the zero value). Also, the range for these values is determined by α but for the width and length of monochromatic matches sharing the upper left corner with one of the areas $A_{i,j}$ (in this case, the range is determined by the width and length of the image). This coding technique is more redundant than others previously designed, but its compression effectiveness is still better than the one of the square greedy matching technique.

3.3 The Parallel Decoder

The parallel decoder has three phases. Observe that at each position of the binary string encoding the image, we read a substring of bits that is either 1111 (recall that the k bits following 1111 provide the area index, where k is the number of bits used to encode it) or can be interpreted as a flag field of a pointer. Then, in the first phase we reduce the binary string to a doubly-linked structure and apply the Euler tour technique in order to identify for each area the corresponding pointers. The reduction works as follows: link each position p of the string to the position next to the end of the substring starting in position p that either is equal to 1111 followed by k bits or can be interpreted as a pointer. For those suffixes of the string which can be interpreted as pointers, their first positions are linked to a special node denoting the end of the coding. For those suffixes of the string which cannot be interpreted as pointers, their first positions are not linked to anything. The linked structure is a forest with one tree rooted in the special node denoting the end of the coding and the other trees rooted in the first position of a suffix of the encoding string not interpretable as a pointer. The first position of the binary string is a leaf of the tree rooted in the special node. The sequence of pointers encoding the image is given by the path from the first position to the root. In order to compute such path we need the children to be doubly-linked to the parent. Then, we need to reserve space for each node to store the links to the children. Each node has at most five children since there are only four different pointer sizes (white, black, raw, or proper match). So, for each position p of the binary sequence we set aside five locations $[p, 1], \ldots, [p, 5]$, initially set to zero. When a link is added from position p' to p, depending on whether the substring starting in position p' is 1111 or can be interpreted as a pointer to a raw, white, black or proper match, the value p' is overwritten on location $[p, 1]$, $[p, 2]$, $[p, 3]$ $[p, 4]$ or $[p, 5]$, respectively. The linking for the substrings starting with 1111 is done first,

since only afterwards we know exactly which substrings can be interpreted as pointers (recall that encoding the width and length of a monochromatic match sharing the left upper corner with one of the areas $A_{i,j}$ depends on the width and length of the whole image). Then, by means of the well-known Euler technique [8] we can linearize the linked structure and apply list ranking to obtain the path from the first position of the sequence to the root of its tree in $O(\alpha)$ time with $O(n/\alpha)$ processors on an exclusive read, exclusive write shared memory machine [1], [4], since the row image size is greater than the size of the encoding binary string. Then, still in $O(\alpha)$ time with $O(n/\alpha)$ processors we can identify the positions on the path corresponding to 1111.

In the second phase of the parallel decoder a different processor decodes the sequence of pointers corresponding to a different area. As far as the pointers to monochromatic matches are considered, each processor decompresses either a match contained in an area or the portion of the match corresponding to the left upper area. Therefore, after the second phase an area might not be decompressed. Obviously, the second phase requires $O(\alpha)$ time and $O(n/\alpha)$ processors.

The third phase completes the decoding. In the previous subsection, we denoted with $A_{i,j}$ for $1 \leq i \leq \lceil m/x \rceil$ and $1 \leq j \leq \lceil m'/y \rceil$ the areas into which the image was partitioned by the encoder. At the first step of the third phase, one processor for each area $A_{2i-1,j}$ decodes $A_{2i,j}$, if $A_{2i-1,j}$ is the upper left portion of a monochromatic match and the length field of the corresponding pointer informs that $A_{2i,j}$ is part of the match. The same is done horizontally for $A_{i,2j-1}$ and $A_{i,2j}$ (using the width field of its pointer) if it is known already by the decoder that $A_{i,2j-1}$ is part of a monochromatic match. Similarly at the k-th step, one processor for each of the areas $A_{(i-1)2^{k-1}+1,j}$, $A_{(i-1)2^{k-1}+2,j}, \ldots, A_{i2^{k-1},j}$, with i odd, decodes the areas $A_{i2^{k-1}+1,j}$, $A_{i2^{k-1}+2,j}, \ldots, A_{(i+1)2^{k-1},j}$, respectively. The same is done horizontally for $A_{i,(j-1)2^{k-1}+1}$, $A_{i,(j-1)2^{k-1}+2}, \ldots, A_{i,j2^{k-1}}$, with j odd, and $A_{i,j2^{k-1}+1}$, $A_{i,j2^{k-1}+2}, \ldots, A_{i,(j+1)2^{k-1}}$. After $O(\log M)$ steps the image is entirely decompressed. Each step takes $O(\alpha)$ time and $O(n/\alpha)$ processors since there is one processor for each area. Therefore, the decoder is realized in $O(\alpha \log M)$ time with $O(n/\alpha)$ processors.

4 The Tree Architecture Implementations

We implement the parallel BLOCK MATCHING encoder and decoder on a full binary tree architecture. We extend the $m \times m'$ image I with dummy rows and columns so that I is partitioned into $x \times y$ areas $A_{i,j}$ for $1 \leq i, j \leq 2^h$, where x and y are $\Theta(\alpha^{1/2})$, $n = mm'$ is the size of the image and $h = \min\{k : 2^k \geq \max\{m/x, m'/y\}\}$. We store these areas into the leaves of the tree according to a one-dimensional layout which allows an easy way of merging the monochromatic ones at the upper levels. Let $\mu = 2^h$. The number of leaves is 2^{2h} and the leaves are numbered from 1 to 2^{2h} from left to right. It follows that the tree has height $2h$. Therefore, the height of the tree is $O(\log n)$ and the number of nodes is $O(n/\alpha)$. Such layout is described by the recursive procedure of figure 3. The initial value for i, j and ℓ is 1 and ℓ is a global variable.

In parallel for each area, each leaf processor applies the sequential parsing algorithm so that in $O(\alpha \log M)$ time each area is parsed into rectangles, some of which are monochromatic. Again, before encoding we wish to compute larger monochromatic rectangles.

STORE(I, μ, i, j)

  if $\mu > 1$

    STORE(I, $\mu/2$, i, j)

    STORE(I, $\mu/2$, $i + \mu/2$, j)

    STORE(I, $\mu/2$, i, $j + \mu/2$)

    STORE(I, $\mu/2$, $i + \mu/2$, $j + \mu/2$)

  else store $A_{i,j}$ into leaf ℓ; $\ell = \ell + 1$

Figure 3. Storing the image into the leaves of the tree.

4.1 Computing the Monochromatic Rectangles

After the compression heuristic has been executed on each area, we have to show how the procedure to compute larger monochromatic rectangles can be implemented on a full binary tree architecture with the same number of processors without slowing it down. This is possible by making some realistic assumptions. Let ℓ_R and w_R be the length and the width of a monochromatic match R, respectively. We define $s_R = \max\{\ell_R, w_R\}$. We make a first assumption that the number of monochromatic matches R with $s_R \geq 2^k \lceil \log^{1/2} n \rceil$ is $O(n/(2^{2k} \log n))$ for $1 \leq k \leq h-1$. While computing larger monochromatic rectangles, we store in each leaf the partial results on the monochromatic rectangles covering the corresponding area (it is enough to store for each rectangle the indices of the areas at the upper left and lower right corners). If i is odd, it follows from the procedure of figure 3 that the processors storing areas $A_{2i-1,j}$ and $A_{2i,j}$ are siblings. Such processors merge $A_{2i-1,j}$ and $A_{2i,j}$ provided they are monochromatic and have the same color by broadcasting the information through their parent. It also follows from such procedure that the same can be done horizontally for $A_{i,2j-1}$ and $A_{i,2j}$ by broadcasting the information through the processors at level $2h-2$. At the k-th step, if areas $A_{(i-1)2^{k-1}+1,j}$, $A_{(i-1)2^{k-1}+2,j}, \ldots, A_{i2^{k-1},j}$, with i odd, were merged for $w_1 \leq j \leq w_2$, the processor storing area $A_{(i-1)2^{k-1}+1,w_1}$ will broadcast to the processors storing the areas $A_{i2^{k-1}+1,j}$, $A_{i2^{k-1}+2,j}, \ldots, A_{(i+1)2^{k-1},j}$ to merge with the above areas for $w_1 \leq j \leq w_2$, if they are monochromatic with the same color. The broadcasting will involve processors up to level $2h-2k+1$. The same is done horizontally, that is, if $A_{i,(j-1)2^{k-1}+1}$, $A_{i,(j-1)2^{k-1}+2}, \ldots, A_{i,j2^{k-1}}$, with j odd, were merged for $\ell_1 \leq i \leq \ell_2$, the processor storing area $A_{\ell_1,(j-1)2^{k-1}+1}$ will broadcast to the processors storing the areas $A_{i,j2^{k-1}+1}$, $A_{i,j2^{k-1}+2}, \ldots, A_{i,(j+1)2^{k-1}}$ to merge with the above areas for $\ell_1 \leq i \leq \ell_2$, if they are monochromatic with the same color. The broadcasting will involve processors up to level $2h-2k$.

After $O(\log M)$ steps, the procedure is completed. If the waste factor is less than 2, as conjectured in [7], we can make a second assumption that each pixel is covered by a constant small number of monochromatic matches. It follows from this second assumption that the information about the monochromatic matches is distributed

among the processors at the same level in a way very close to uniform. Then, it follows from the first assumption that the amount of information each processor of the tree must broadcast is constant. Therefore, each step takes $O(\alpha)$ time and the image parsing phase is realized with $O(\alpha \log M)$ time and $O(n/\alpha)$ processors.

4.2 The Parallel Encoder

The sequence of pointers is trivially produced by the processors which are leaves of the tree. For the monochromatic rectangles, the pointer is written in the leaf storing the area at the upper left corner. Differently from the shared memory machine decoder, the order of the pointers is the one of the leaves. Since some areas could be entirely covered by a monochromatic match, the subsequence of pointers corresponding to a given area is ended with 1111 followed by the index of the leaf storing the next area to decode. We define a variable for each leaf. This variable is set to 1 if the leaf stores at least a pointer, 0 otherwise. Then, the index of the next area to decode is computed for each leaf by parallel suffix computation. Moreover, with the possibility of a parallel output the sequence can be put together by parallel prefix. This is, obviously, realized in $O(\alpha)$ time with $O(n/\alpha)$ processors.

4.3 The Parallel Decoder

We store each encoding of an area in a leaf of the tree. The storing procedure reads the encoding binary string from left to right. When it finds the substring 1111, this denotes the end of the encoding of an area and the next k bits provide the leaf index of the next area where k is the number of bits used to encode it. At this point, each processor at the leaf level completes the second phase of the decoder described in subsection 3.3. Then, the third and last phase of the shared memory machine decoder has the same parallel computational complexity on the tree architecture with the same realistic assumptions we made for the coding phase. In conclusion, the decoder takes $O(\alpha \log M)$ time on a full binary tree architecture with $O(n/\alpha)$ processors.

5 Conclusions

Parallel coding and decoding algorithms for lossless image compression by block matching were shown requiring $O(\log M \log n)$ time and $O(n/\log n)$ processors on a full binary tree architecture, where n is the size of the image and M is the size of the match. The parallel coding algorithm is work-optimal since the sequential time required by the coding is $\Omega(n \log M)$. On the other hand, the parallel decoding algorithm is not work-optimal since the sequential decompression time is linear. These implementations have the same performance of the shared memory machine algorithms under some realistic assumptions and if we do not consider the input/output process. In [3], with such assumptions these algorithms had been realized on a multigrid which is the simplest architecture among the ones with small diameter and large bisection width. Such realistic assumptions are that each pixel is covered by a small constant number of monochromatic rectangles of the image parsing and that the increasing of the dimensions of the monochromatic rectangles is proportional to the decreasing of the number of monochromatic rectangles with such dimensions. We have shown in this paper that with the assumptions made for the multigrid we can relax on the requirement of an architecture with large bisection width and design compression

and decompression on a two-dimensional architecture such as a full binary tree. The communication cost might be low enough to realize efficient implementations on one of the available parallel machines since the algorithms are scalable.

References

1. R. P. Brent: *The parallel evaluation of general arithmetic expressions.* Journal of the ACM, 21 1974, pp. 201–206.
2. R. P. Brent: *A linear algorithm for data compression.* Australian Computer Journal, 19 1987, pp. 64–68.
3. L. Cinque and S. DeAgostino: *Lossless image compression by block matching on practical massively parallel architectures*, in Proceedings Prague Stringology Conference, 2008, pp. 26–34.
4. R. Cole and U. Vishkin: *Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time.* SIAM Journal on Computing, 17 1988, pp. 148–152.
5. A. Lempel and J. Ziv: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, 23 1977, pp. 337–343.
6. J. A. Storer: *Lossless image compression using generalized LZ1-type methods*, in Proceedings IEEE Data Compression Conference, 1996, pp. 290–299.
7. J. A. Storer and H. Helfgott: *Lossless image compression by block matching.* The Computer Journal, 40 1997, pp. 137–145.
8. R. E. Tarjan and U. Vishkin: *An efficient parallel biconnectivity algorithm.* SIAM Journal on Computing, 14 1985, pp. 862–874.
9. J. R. Waterworth: *Data compression system.* US Patent 4 701 745, 1987.
10. D. A. Whiting, G. A. George, and G. E. Ivey: *Data compression apparatus and method.* US Patent 5016009, 1991.