Algorithms **2011**, *4*, 183-199; doi:10.3390/a4030183

OPEN ACCESS

algorithms

ISSN 1999-4893

www.mdpi.com/journal/algorithms

Article

Lempel–Ziv Data Compression on Parallel and Distributed Systems

Sergio De Agostino

Computer Science Department, Sapienza University, Via Salaria 113, Rome, Italy;
E-Mail: eagostino@di.uniroma1.it; Tel.: +39-06-4991-8529; Fax: +39-06-857-1842

Received: 29 July 2011; in revised form: 23 August 2011 / Accepted: 30 August 2011 / Published: 14 September 2011

Abstract: We present a survey of results concerning Lempel–Ziv data compression on parallel and distributed systems, starting from the theoretical approach to parallel time complexity to conclude with the practical goal of designing distributed algorithms with low communication cost. Storer's extension for image compression is also discussed.

Keywords: dictionary-based compression; string factorization; parallel complexity; distributed algorithm; binary image

---

1. Introduction

Lempel–Ziv compression [1–3] is based on string factorization. Two different factorization processes exist with no memory constraints. With the first one (LZ1) [2], each factor is independent from the others since it extends by one character the longest match with a substring to its left in the input string. With the second one (LZ2) [3], each factor is instead the extension by one character of the longest match with one of the previous factors. This computational difference implies that while LZ1 compression has efficient parallel algorithms [4,5] LZ2 compression is hard to parallelize [6]. This difference is maintained when bounded memory versions of Lempel–Ziv compression are considered [5,7,8]. On the other hand, parallel decompression is possible for both approaches [9]. Moreover, distributed algorithms for the LZ1 and LZ2 methods approximating in practice their compression effectiveness have been realized [8,10–12]. The LZ2 method is less effective than LZ1 but it is more scalable if we use a tree architecture. In Section 2, we describe the Lempel–Ziv compression techniques and in Section 3, we present the bounded memory versions. In Sections 4 and and 5 we discuss how Lempel–Ziv data

compression and decompression can be implemented on parallel and distributed systems, respectively. The Storer's extension [13] for image compression is discussed in Section 6. Conclusions and future work are given in Section 7.

2. Lempel–Ziv Data Compression

Lempel–Ziv compression is a dictionary-based technique. In fact, the factors of the string are substituted by *pointers* to copies stored in a dictionary which are called *targets*. LZ1 (LZ2) compression is also called the sliding (dynamic) dictionary method.

2.1. *LZ1 Compression*

Given an alphabet A and a string S in A^*, the LZ1 factorization of S is $S = f_1 f_2 \cdots f_i \cdots f_k$ where f_i is the shortest substring which does not occur previously in the prefix $f_1 f_2 \cdots f_i$ for $1 \leq i \leq k$. With such factorization, the encoding of each factor leaves one character uncompressed. To avoid this, a different factorization was introduced (LZSS factorization) where f_i is the longest match with a substring occurring in the prefix $f_1 f_2 \cdots f_i$ if $f_i \neq \lambda$, otherwise f_i is the alphabet character next to $f_1 f_2 \cdots f_{i-1}$ [15]. f_i is encoded by the pointer $q_i = (d_i, \ell_i)$, where d_i is the displacement back to the copy of the factor and ℓ_i is the length of the factor (LZSS compression). If $d_i = 0$, l_i is the alphabet character. In other words the dictionary is defined by a window sliding its right end over the input string, that is, it comprises all the substrings of the prefix read so far in the computation. It follows that the dictionary is both *prefix* and *suffix* since all the prefixes and suffixes of a dictionary element are dictionary elements. The position of the longest match in the prefix with the current position can be computed in real time by means of a suffix tree data structure [16,17].

2.2. *LZ2 Compression*

The LZ2 factorization of a string S is $S = f_1 f_2 \cdots f_i \cdots f_k$ where f_i is the shortest substring which is different from one of the previous factors. As for LZ1 the encoding of each factor leaves one character uncompressed. To avoid this a different factorization was introduced (LZW factorization) where each factor f_i is the longest match with the concatenation of a previous factor and the next character [18]. f_i is encoded by a pointer q_i to such concatenation (LZW compression). LZ2 and LZW compression can be implemented in real time by storing the dictionary with a trie data structure. Differently from LZ1 and LZSS, the dictionary is only prefix.

2.3. *Greedy* versus *Optimal Factorization*

The pointer encoding the factor f_i has a size increasing with the index i. This means that the lower is the number of factors for a string of a given length, the better is the compression. The factorizations described in the previous subsections are produced by greedy algorithms. The question is whether the greedy approach is always optimal, that is, if we relax the assumption that each factor is the longest match, can we do better than greedy? The answer is negative with suffix dictionaries as for LZ1 or LZSS compression. On the other hand, the greedy approach is not optimal for LZ2 or LZW compression.

However, the optimal approach is NP-complete [19] and the greedy algorithm approximates with an $O(n^{\frac{1}{4}})$ multiplicative factor the optimal solution [20].

3. Bounded Size Dictionary Compression

The factorization processes described in the previous section are such that the number of different factors (that is, the dictionary size) grows with the string length. In practical implementations instead the dictionary size is bounded by a constant and the pointers have equal size. While for LZSS (or LZ1) compression this can be simply obtained by sliding a fixed length window and by bounding the match length, for LZW (or LZ2) compression dictionary elements are removed by using a deletion heuristic. The deletion heuristics we describe in this section are FREEZE, RESTART and LRU [21]. Then, we give more details on sliding window compression.

3.1. The Deletion Heuristics

Let $d+\alpha$ be the cardinality of the fixed size dictionary where α is the cardinality of the alphabet. With the FREEZE deletion heuristic, there is a first phase of the factorization process where the dictionary is filled up and "frozen". Afterwards, the factorization continues in a "static" way using the factors of the frozen dictionary. In other words, the LZW factorization of a string S using the FREEZE deletion heuristic is $S = f_1 f_2 \cdots f_i \cdots f_k$ where f_i is the longest match with the concatenation of a previous factor f_j, with $j \leq d$, and the next character. The shortcoming of this heuristic is that after processing the string for a while the dictionary often becomes obsolete. A more sophisticated deletion heuristic is RESTART, which monitors the compression ratio achieved on the portion of the input string read so far and, when it starts deteriorating, restarts the factorization process. Let $f_1 f_2 \cdots f_j \cdots f_i \cdots f_k$ be such factorization with j the highest index less than i where the restart operation happens. Then, f_j is an alphabet character and f_i is the longest match with the concatenation of a previous factor f_h, with $h \geq j$, and the next character (the restart operation removes all the elements from the dictionary but the alphabet characters). This heuristic is used by the Unix command Compress since it has a good compression effectiveness and it is easy to implement. However, the best deletion heuristic is LRU (last recently used strategy). The LRU deletion heuristic removes elements from the dictionary in a continuous way by deleting at each step of the factorization the least recently used factor which is not a proper prefix of another one.

3.2. Compression with Finite Windows

As mentioned at the beginning of this section, LZSS (or LZ1) bounded size dictionary compression is obtained by sliding a fixed length window and by bounding the match length. A real time implementation of compression with finite window is possible using a suffix tree data structure [22]. Much simpler real time implementations are realized by means of hashing techniques providing a specific position in the window where a good approximation of the longest match is found on realistic data. In [23], the three current characters are hashed to yield a pointer into the already compressed text. In [24], hashing of strings of all lengths is used to find a match. In both methods, collisions are resolved by overwriting. In [25], the two current characters are hashed and collisions are chained via an offset array. Also the Unix gzip compressor chains collisions but hashes three characters [26].

3.3. Greedy versus *Optimal Factorization*

Greedy factorization is optimal for compression with finite windows since the dictionary is suffix. With LZW compression, after we fill up the dictionary using the FREEZE or RESTART heuristic, the greedy factorization we compute with such dictionary is not optimal since the dictionary is not suffix. However, there is an optimal semi-greedy factorization which is computed by the procedure of figure 1 [27,28]. At each step, we select a factor such that the longest match in the next position with a dictionary element ends to the rightest. Since the dictionary is prefix, the factorization is optimal. The algorithm can even be implemented in real time with a modified suffix tree data structure [27].

Figure 1. The semi-greedy factorization procedure.

j:=0; i:=0
repeat forever
  for $k = j + 1$ **to** $i + 1$ **compute**
    $h(k)$: $x_k ... x_{h(k)}$ is the longest match in the k^{th} position
  let k' be such that $h(k')$ is maximum
  $x_j ... x_{k'-1}$ is a factor of the parsing; $j := k'$; $i := h(k')$

4. Lempel–Ziv Compression on a Parallel System

LZSS (or LZ1) compression can be efficiently parallelized on a PRAM EREW [4,5,8], that is, a parallel machine where processors access a shared memory without reading and writing conflicts. On the other hand, LZW (or LZ2) compression is P-complete [6] and, therefore, hard to parallelize. Decompression, instead, is parallelizable for both methods [9]. As far as bounded size dictionary compression is concerned, the "parallel computation thesis" claims that sequential work space and parallel running time have the same order of magnitude giving theoretical underpinning to the realization of parallel algorithms for LZW compression using a deletion heuristic. However, the thesis concerns unbounded parallelism and a practical requirement for the design of a parallel algorithm is a limited number of processors. A stronger statement is that sequential logarithmic work space corresponds to parallel logarithmic running time with a polynomial number of processors. Therefore, a fixed size dictionary implies a parallel algorithm for LZW compression satisfying these constraints. Realistically, the satisfaction of these requirements is a necessary but not a sufficient condition for a practical parallel algorithm since the number of processors should be linear, which does not seem possible for the RESTART deletion heuristic. Moreover, the SC^k-completeness of LZ2 compression using the LRU deletion heuristic and a dictionary of polylogarithmic size shows that it is unlikely to have a parallel complexity involving reasonable multiplicative constants [7]. In conclusion, the only practical LZW compression algorithm for a shared memory parallel system is the one using the FREEZE deletion heuristic. We will see these arguments more in details in the next subsections.

4.1. Sliding Window Compression on a Parallel System

We present compression algorithms for sliding dictionaries on an exclusive read, exclusive write shared memory machine requiring $O(k)$ time with $O(n/k)$ processors if k is $\Omega(\log n)$, with the practical and realistic assumption that the dictionary size and the match length are constant [8]. As previously

mentioned, greedy factorization is optimal with sliding dictionaries. In order to compute a greedy factorization in parallel we find the greedy match in each position i of the input string and link i to $j + 1$, where j is the last position of the match. If the greedy match ends the string i is linked to $n + 1$, where n is the length of the string. It follows that we obtain a tree rooted in $n + 1$ and the positions of the factors are given by the path from 1 to $n + 1$. Such tree can be built in $\mathrm{O}(k)$ time with $\mathrm{O}(n/k)$ processors. In fact, on each block of k positions one processor has to compute a match having constant length and the reading conflicts with other processors are solved in logarithmic time by standard broadcasting techniques. Then, since for each node of the tree the number of children is bounded by the constant match length it is easy to add the links from a parent node to its children in $\mathrm{O}(k)$ time with $\mathrm{O}(n/k)$ processors and apply the well-known Euler tour technique to this doubly linked tree structure to compute the path from 1 to $n + 1$.

4.2. The Completeness Results

NC is the class of problems solvable with a polynomial number of processors in polylogarithmic time on a parallel random access machine and it is conjectured to be a proper subset of P, the class of problems solvable in sequential polynomial time. LZ2 and LZW compression with an unbounded dictionary have been proved to be P-complete [6] and, therefore, hard to parallelize. SC is the class of problems solvable in polylogarithmic space and sequential polynomial time. The LZ2 algorithm with LRU deletion heuristic on a dictionary of size $\mathrm{O}(\log^k n)$ can be performed in polynomial time and $\mathrm{O}(\log^k n \log\log n)$ space, where n is the length of the input string. In fact, the trie requires $\mathrm{O}(\log^k n)$ space by using an array implementation since the number of children for each node is bounded by the alphabet cardinality. The $\log\log n$ factor is required to store the information needed for the LRU deletion heuristic since each node must have a different age, which is an integer value between 0 and the dictionary size. Obviously, this is true for the LZW algorithm as well. If the size of the dictionary is $\mathrm{O}(\log^k n)$, the LRU strategy is log-space hard for SC^k, the class of problems solvable simultaneously in polynomial time and $\mathrm{O}(\log^k n)$ space [7]. The problem belongs to SC^{k+1}. This hardness result is not so relevant for the space complexity analysis since $\Omega(\log^k n)$ is an obvious lower bound to the work space needed for the computation. Much more interesting is what can be said about the parallel complexity analysis. In [7] it was shown that LZ2 (or LZW) compression using the LRU deletion heuristic with a dictionary of size c can be performed in parallel either in $\mathrm{O}(\log n)$ time with $2^{\mathrm{O}(c \log c)} n$ processors or in $2^{\mathrm{O}(c \log c)} \log n$ time with $\mathrm{O}(n)$ processors. This means that if the dictionary size is constant, the compression problem belongs to NC. NC and SC are classes that can be viewed in some sense symmetric and are believed to be incomparable. Since log-space reductions are in NC, the compression problem cannot belong to NC when the dictionary size is polylogarithmic if NC and SC are incomparable. We want to point out that the dictionary size c figures as an exponent in the parallel complexity of the problem. This is not by accident. If we believe that SC is not included in NC, then the SC^k-hardness of the problem when c is $\mathrm{O}(\log^k n)$ implies the exponentiation of some increasing and diverging function of c. In fact, without such exponentiation either in the number of processors or in the parallel running time, the problem would be SC^k-hard and in NC when c is $\mathrm{O}(\log^k n)$. Observe that the P-completeness of the problem, which requires a superpolylogarithmic value for c, does not suffice to infer this exponentiation since c can figure as a multiplicative factor of the time function. Moreover, this is a unique case so far where somehow we use

hardness results to argue that practical algorithms of a certain kind (NC in this case) do not exist because of huge multiplicative constant factors occurring in their analysis. In [7] a relaxed version (RLRU) was introduced which turned out to be the first (and only so far) natural SC^k-complete problem. RLRU partitions the dictionary in p equivalence classes, so that all the elements in each class are considered to have the same "age" for the LRU strategy. RLRU turns out to be as good as LRU even when p is equal to 2 [29]. Since RLRU removes an arbitrary element from the equivalence class with the "older" elements, the two classes (when p is equal to 2) can be implemented with a couple of stacks, which makes RLRU slightly easier to implement than LRU in addition to be more space efficient.

4.3. LZW Compression on a Parallel System

As mentioned at the beginning of this section, the only practical LZW compression algorithm for a shared memory parallel system is the one using the FREEZE deletion heuristic. After the dictionary is built and frozen, a parallel procedure similar to the one for LZ1 compression is run. To compute a greedy factorization in parallel we find the greedy match with the frozen dictionary in each position i of the input string and link i to $j+1$, where j is the last position of the match. If the greedy match ends the string i is linked to $n+1$, where n is the length of the string. It follows that we obtain a tree rooted in $n+1$ and the positions of the factors of the greedy parsing are given by the path from 1 to $n+1$. In order to compute an optimal factorization we parallelize the semi-greedy procedure. The longest sequence of two matches in each position i of the string can be computed in $O(k)$ time with $O(n/k)$ processors, in a similar way as for the greedy procedure. Then, position i is linked to the position of the second match. If the second match is empty, i is linked to $n+1$. Again, we obtain a tree rooted in $n+1$ and the positions of the factors are given by the path from 1 to $n+1$. The tree and the path are computed in $O(k)$ time with $O(n/k)$ processors if k is $\Omega(\log n)$, as in the first subsection without reading and writing conflicts [8]. The parallelization of the sequential LZW compression algorithm with the RESTART deletion heuristic is not practical enough since it requires a quadratic number of processors [7].

4.4. Parallel Decompression

The design of parallel decoders is based on the fact that the Euler tour technique can also be used to find the trees of a forest in $O(k)$ time with $O(n/k)$ processors on a shared memory parallel machine without writing and reading conflicts, if k is $\Omega(\log n)$ and n is the number of nodes. We present decoders paired with the practical coding implementations using bounded size dictionaries. First, we see how to decode the sequence of pointers $q_i = (d_i, \ell_i)$ produced by the LZSS method with $1 \leq i \leq m$ [9]. If $s_1, ..., s_m$ are the partial sums of $l_1, ..., l_m$, the target of q_i encodes the substring over the positions $s_{i-1}+1 \cdots s_i$ of the output string. Link the positions $s_{i-1}+1 \cdots s_i$ to the positions $s_{i-1}+1-d_i \cdots s_{i-1}+1-d_i+l_i-1$, respectively. If $d_i = 0$, the target of q_i is an alphabet character and the corresponding position in the output string is not linked to anything. Therefore, we obtain a forest where all the nodes in a tree correspond to positions of the decoded string where the character is represented by the root. The reduction from the decoding problem to the problem of finding the trees in a forest can be computed in $O(k)$ time with $O(n/k)$ processors where n is the length of the output string, because this is the complexity of computing the partial sums since $m \leq n$. Afterwards, one processor stores the parent

pointers in an array of size n for a block of k positions. We can make the forest a doubly linked structure since the window size is constant and apply the Euler tour technique to find the trees.

With LZW compression using the FREEZE deletion heuristic the parallel decoder is trivial. We wish to point out that the decoding problem is interesting independently from the computational efficiency of the encoder. In fact, in the case of compressed files stored in a ROM only the computational efficiency of decompression is relevant. With the RESTART deletion heuristic, a special mark occurs in the sequence of pointers each time the dictionary is cleared out so that the decoder does not have to monitor the compression ratio. The positions of the special mark are detected by parallel prefix. Each subsequence $q_1 \cdots q_m$ of pointers between two consecutive marks can be decoded in parallel but the pointers do not contain the information on the length of their targets and it has to be computed. The target of the pointer q_i in the subsequence is the concatenation of the target of the pointer in position $q_i - \alpha$ with the first character of the target of the pointer in position $q_i - \alpha + 1$, where α is the alphabet cardinality. Then, in parallel for each i, link pointer q_i to the pointer in position $q_i - \alpha$, if $q_i > \alpha$. Again, we obtain a forest where each tree is rooted in a pointer representing an alphabet character and the length l_i of the target of a pointer q_i is equal to the level of the pointer in the tree plus 1. It is known from [1] that the largest number of distinct factors whose concatenation forms a given string of length ℓ is $\mathrm{O}(\ell/\log \ell)$. Since a factor of the LZW factorization of a string appears a number of times which is at most equal to the alphabet cardinality, it follows that m is $\mathrm{O}(\ell/\log \ell)$ if ℓ is the length of the substring encoded by the subsequence $q_1 \cdots q_m$. Then, building such a forest takes $\mathrm{O}(k)$ time with $\mathrm{O}(n/k)$ processors on a shared memory parallel machine without writing and reading conflicts if k is $\Omega(\log n)$. By means of the Euler tour technique, we can compute the trees of such forest and the level of each node in its own tree in $\mathrm{O}(k)$ time with $\mathrm{O}(n/k)$. Therefore, we can compute the lengths $l_1, ..., l_m$ of the targets. If $s_1, ..., s_m$ are the partial sums, the target of q_i is the substring over the positions $s_{i-1} + 1 \cdots s_i$ of the output string. For each q_i which does not correspond to an alphabet character, define $first(i) = s_{q_i - \alpha - 1} + 1$ and $last(i) = s_{q_i - \alpha} + 1$. Since the target of the pointer q_i is the concatenation of the target of the pointer in position $q_i - \alpha$ with the first character of the target of the pointer in position $q_i - \alpha + 1$, link the positions $s_{i-1} + 1 \cdots s_i$ to the positions $s_{first(i)} \cdots s_{last(i)}$, respectively. As in the sliding dictionary case, if the target of q_i is an alphabet character the corresponding position in the output string is the root of a tree in a forest and all the nodes in a tree correspond to positions of the decoded string where the character is the root. Since the number of children for each node is at most α, in $\mathrm{O}(k)$ time and $\mathrm{O}(n/k)$ processors we can store the forest in a doubly linked structure and decode by means of the Euler tour technique [9]. Las Vegas work-optimal parallel decoders for LZ1 and LZ2 compression were presented in [30,31] with a computational complexity independent from the dictionary size and the match length.

5. Lempel–Ziv Compression on a Distributed System

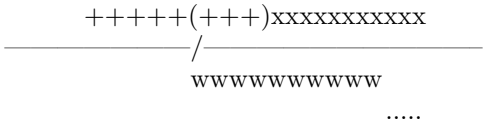
Distributed systems have two types of complexity, the interprocessor communication and the input-output mechanism. While the input/output issue is inherent to any parallel algorithm and has standard solutions, the communication cost of the computational phase after the distribution of the data among the processors and before the output of the final result is obviously algorithm-dependent. So, we need to limit the interprocessor communication and involve more local computation to design a practical algorithm. The simplest model for this phase is, of course, a simple array of processors

with no interconnections and, therefore, no communication cost. We describe on such model for every integer k greater than 1 an O(kw) time, O(n/kw) processors distributed algorithm factorizing an input string S with a cost which approximates the cost of the LZSS factorization within the multiplicative factor $(k + m - 1)/k$, where n, m and w are the lengths of the input string, the longest factor and the window respectively [8]. As far as LZW compression is concerned, if we use a RESTART deletion heuristic clearing out the dictionary every ℓ characters of the input string we can trivially parallelize the factorization process with an O(ℓ) time,O(n/ℓ) processors distributed algorithm. We also present on a tree architecture an algorithm which in time O(km) with O(n/km) processors is guaranteed to produce a factorization of S with a cost approximating the cost of the optimal factorization within the multiplicative factor $(k + 1)/k$ [10,12]. These algorithms provide approximation schemes for the corresponding factorization problems since the multiplicative approximation factors converge to 1 when km and kw converge to ℓ and to n, respectively.

5.1. Sliding Window Compression on a Distributed System

We simply apply in parallel sliding window compression to blocks of length kw. It follows that the algorithm requires O(kw) time with n/kw processors and the multiplicative approximation factor is $(k + m - 1))/k$ with respect to any parsing. In fact, the number of factors of an optimal (greedy) factorization on a block is at least kw/m while the number of factors of the factorization produced by the scheme is at most $(k-1)w/m+w$. As shown in Figure 2, the boundary might cut a factor (sequence of plus signs) and the length w of the initial full size window of the block (sequence of w's) is the upper bound to the factors produced by the scheme in it. Yet, the factor cut by the boundary might be followed by another factor (sequence of x's) which covers the remaining part of the initial window. If this second factor has a suffix to the right of the window, this suffix must be a factor of the sliding dictionary defined by it (dotted line) and the multiplicative approximation factor follows.

Figure 2. The making of the surplus factors.



We obtain an approximation scheme which is suitable for a small scale system but due to its adaptiveness it works on a large scale parallel system when the file size is large. From a practical point of view, we can apply something like the gzip procedure to a small number of input data blocks achieving a satisfying degree of compression effectiveness and obtaining the expected speed-up on a real parallel machine. Making the order of magnitude of the block length greater than the one of the window length largely beats the worst case bound on realistic data. The window length is usually several thousands kilobytes. The compression tools of the Zip family, as the Unix command "gzip" for example, use a window size of at least 32 K. It follows that the block length in our parallel implementation should be about 300 K and the file size should be about one third of the number of processors in megabytes. An approach using a tree architecture slightly improves compression effectiveness [11]. However, the

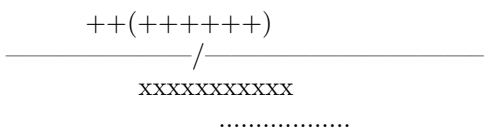
scalability of a parallel implementation of sliding window compression on a distributed system with low communication cost seems to guarantee robustness only on very large files. We show in the next subsection that LZW compression is scalable and robust on arbitrary files if implemented on a tree architecture [12].

5.2. LZW Compression on a Distributed System

As mentioned at the beginning of this section, if we use a RESTART deletion heuristic clearing out the dictionary every ℓ characters of the input string we can trivially parallelize the factorization process with an $O(\ell)$ time, $O(n/\ell)$ processors distributed algorithm. LZW compression with the RESTART deletion heuristic was initially presented in [18] with a dictionary of size 2^{12} and is employed by the Unix command "compress" with a dictionary of size 2^{16}. Therefore, in order to have a satisfying compression effectiveness the distributed algorithm might work with blocks of length ℓ even greater than 300K on realistic data. After a dictionary is filled up for each block though, the factorization of the remaining suffix of the block can be approximated within the multiplicative factor $(k+1)/k$ in time $O(km)$ with $O(n/km)$ processors on a tree architecture. Every leaf processor stores a sub-block of length $m(k+2)$ and a copy of the dictionary which are broadcasted from some level of the tree where the first phase of the computation has been executed. Adjacent sub-blocks overlap on $2m$ characters. We call a *boundary match* a factor covering positions of two adjacent sub-blocks. We execute the following algorithm:

- for each block, every processor but the one associated with the last sub-block computes the boundary match between its sub-block and the next one which ends furthest to the right;
- each processor computes the optimal factorization from the beginning of the boundary match on the left boundary of its sub-block to the beginning of the boundary match on the right boundary.

Figure 3. The making of a surplus factor.



Stopping the factorization of each sub-block at the beginning of the right boundary match might cause the making of a surplus factor, which determines the multiplicative approximation factor $(k+1)/k$ with respect to any factorization. In fact, as it is shown in Figure 3, the factor in front of the right boundary match (sequence of x's) might be extended to be a boundary match itself (sequence of plus signs) and to cover the first position of the factor after the boundary (dotted line). In [10], it is shown experimentally that for $k = 10$ the compression ratio achieved by such factorization is about the same as the sequential one. Then, compression can be effective on a large scale system even if the size of the file is not large.

5.3. Decompression on a Distributed System

To decode the compressed files on a distributed system, it is enough to use a special mark occurring in the sequence of pointers each time the coding of a block ends. The input phase distributes the subsequences of pointers coding each block among the processors. If the file is encoded by an LZ2 compressor implemented on a large scale tree architecture, a second special mark indicates for each block the end of the coding of a sub-block and the coding of each block is stored at the same level of the tree. The first sub-block for each block is decoded by one processor to learn the corresponding dictionary. Then, the subsequences of pointers coding the sub-blocks are broadcasted to the leaf processors with the corresponding dictionary.

6. Image Compression

The best lossless image compressors are enabled by arithmetic encoders based on the model driven method [32]. The *model driven method* consists of two distinct and independent phases: *modeling* [33] and *coding* [34]. Arithmetic encoders are the best model driven compressors both for binary images (JBIG) [35] and for grey scale and color images (CALIC) [36], but they are often ruled out because they are too complex. As far as the model driven method for grey scale and color image compression is concerned, the modeling phase consists of three components: the determination of the context of the next pixel, the prediction of the next pixel and a probabilistic model for the *prediction residual*, which is the value difference between the actual pixel and the predicted one. In the coding phase, the prediction residuals are encoded. The use of prediction residuals for grey scale and color image compression relies on the fact that most of the times there are minimal variations of color in the neighborhood of one pixel. LOCO-I (JPEG-LS) [37] is the current lossless standard in low-complexity applications and involves Golomb-Rice codes [38,39] rather than the arithmetic ones. A low-complexity application compressing 8x8 blocks of a grey-scale or color image by means of a header and a fixed-length code is PALIC [40] which can be implemented on an arbitrarily large scale system at no communication cost. As explained in [40], parallel implementations of LOCO-I would require more sophisticated architectures than a simple array of processors. PALIC achieves 80 percent of the compression obtained with LOCO-I and is an extremely local procedure which is able to achieve a satisfying degree of compression by working independently on different very small blocks. On the other hand, no perfectly scalable low-complexity binary image compressor has been designed so far on an array architecture with no interprocessor communication. BLOCK MATCHING [13,14] is the best low-complexity compressor for binary images and extends data compression via textual substitution to two-dimensional data by compressing sub-images rather than substrings [2,15], achieving 80 percent of the compression obtained with JBIG. However, it does not work locally since it applies a generalized LZ1-type method with an unrestricted window. Storer extended the LZ1 method to binary image lossless compression by means of a square block matching technique [13]. A slower but more effective binary image compressor employs rectangular matches [14] and it is more suitable for scalable parallel and distributed implementations [8].

6.1. The Generalized LZ1-Type Method

The square matching compression procedure scans an $m \times m'$ image by a raster scan (the greedy matching technique could work with any other scan described in [14]). A 64K table with one position for each possible 4×4 subarray is the only data structure used. All-zero and all-one squares are handled differently. The encoding scheme is to precede each item with a flag field indicating whether there is a monochromatic square, a match, or raw data. When there is a match, the 4×4 subarray in the current position (i, j) is hashed to yield a pointer to a copy in some position (k, h). This pointer is used for the current greedy match and then replaced in the hash table by a pointer to the current position. The procedure for computing the largest square match takes O(M) time, where M is the size of the match. Obviously, this procedure can be used to compute the largest monochromatic square 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 square in that position. Otherwise, we compute the largest 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 by matches are skipped in the linear scan of the image. We wish to point out that besides the proper matches we call a match every square of the parsing of the image produced by the heuristic. We also call pointer the encoding of a match. Each pointer starts with a flag field indicating whether there is a monochromatic match (0 for the white ones and 10 for the black ones), a proper match (110) or a raw match (111). If we use rectangular matches, the pointer is more expensive in size since two fields for the dimension of proper or monochromatic matches are needed. However, the matches are larger and there is a gain in compression effectiveness. The procedure for finding the largest rectangular match is described in Figure 4. We denote with $p_{i,j}$ the pixel in position (i, j). 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 $1 \times l$ computed at the first step is the current rectangular match and the sizes of its sides are stored in $side1$ and $side2$. 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 5 we apply the procedure to find the largest rectangular match between position $(0, 0)$ and $(6, 6)$. 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. For typical bi-level images this scheme is extremely fast for

square matches and there is no significant slowdown over simply reading and writing the image. On the other hand, the procedure for computing the largest rectangular match takes $O(M \log M)$ time. In fact, in the worst case a rectangle of size M could be detected on row i, a rectangle of size $M/2$ on row $i+1$, a rectangle of size $M/3$ on row $i+2$ and so on. Therefore, the running time of the compression heuristic is $\Omega(n)$ or $\Omega(n \log M)$ for square or rectangular matches respectively, where n is the size of the image. The analysis of the running time of such generalized LZ1-type method involves a so called *waste factor*, which is the average number of matches in the parsing of the image covering the same pixel. It is conjectured by Storer that this ia always less than 2 on realistic image data. Therefore, the square and rectangular block matching heuristics have a running time equal to $O(n)$ and $O(n \log M)$.

Figure 4. Computing the largest rectangle match in (i, j) and (k, h).

$w = k;$
$r = i;$
$width = m;$
$length = 0;$
$side1 = side2 = area = 0;$
repeat
    Let $p_{r,j} \cdots p_{r,j+\ell-1}$ be the longest match in (w, h) with $\ell \leq width;$
    $length = length + 1;$
    $width = \ell;$
    $r = r + 1;$
    $w = w + 1;$
    if $(length * width > area)$ {
        $area = length * width;$
        $side1 = length;$
        $side2 = width;$
    }
until $area \geq width * (i - k + 1)$ or $w = i + 1$

6.2. The Implementation on a Parallel System

Compression and decompression parallel implementations are described, requiring $O(\alpha \log M)$ time with $O(n/\alpha)$ processors for any integer square value $\alpha \in \Omega(\log n)$ on a PRAM EREW [8]. 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 rectangular block matching algorithm so that in $O(\alpha \log M)$ time each area is parsed into rectangles, some of which are monochromatic. In the theoretical context of unbounded parallelism, α can be arbitrarily small and before encoding we wish to compute larger monochromatic rectangles. The monochromatic rectangles are computed by merging adjacent monochromatic areas without considering those monochromatic matches properly contained in some area. 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 $A_{(i-1)2^{k-1}+1,j}$, $A_{(i-1)2^{k-1}+2,j}$, $\cdots$ $A_{i2^{k-1},j}$, with i odd, were merged, then they will merge with areas $A_{i2^{k-1}+1,j}$, $A_{i2^{k-1}+2,j}$, $\cdots$ $A_{(i+1)2^{k-1},j}$, if they are monochromatic with the same color.

The same is done horizontally for $A_{i,(j-1)2^{k-1}+1}$, $A_{i,(j-1)2^{k-1}+2}$, $\cdots A_{i,j2^{k-1}}$, with j odd, and $A_{i,j2^{k-1}+1}$, $A_{i,j2^{k-1}+2}$, $\cdots A_{i,(j+1)2^{k-1}}$. After $\mathrm{O}(\log M)$ steps, the procedure is completed and each step takes $\mathrm{O}(\alpha)$ time and $\mathrm{O}(n/\alpha)$ processors since there is one processor for each area. Therefore, the image parsing phase is realized in $\mathrm{O}(\alpha \log M)$ time with $\mathrm{O}(n/\alpha)$ processors on an exclusive read, exclusive write shared memory machine. In order to implement parallel decompression, we need to indicate the end of the encoding of a specific area. So, we change the encoding scheme by associating the flag field 1110 to the raw match so that 1111 could indicate the end of the sequence of pointers corresponding to a given area. Then, the flag field 1110 is followed by sixteen bits uncompressed, flag fields 0 and 10 by the width and the length of the rectangle while flag field 110 is also followed by the position of the matching copy. The values following the flag fields are represented by a fixed length code. Since some areas could be entirely covered by a monochromatic rectangle 1111 is followed by the index associated with the next area by the raster scan. The interested reader can see how the coding and decoding phase are designed in [8].

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

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	

6.3. The Implementation on a Distributed System

An implementation on a small scale system with no interprocessor communication was realized since we were able to partition an image into up to a hundred areas and to apply the square block matching heuristic independently to each area with no relevant loss of compression effectiveness. In [8] we obtained the expected speed-up on the compression and decompression times doubling up the number of processors of a 256 Intel Xeon 3.06 GHz processors machine (avogadro.cilea.it) from 1 to 32. Working only with monochromatic matches and using a variable length encoding for the pointer fields after the flag, we were even able to preserve the compression effectiveness on a large scale system with a thousand processor and no interprocessor communication [41]. To obtain perfect scalability, however, we implemented in [8] the PRAM EREW procedure on a full binary tree architecture under some realistic

assumptions on the image data as follows. 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 $\mathrm{O}(\log n)$ and the number of nodes is $\mathrm{O}(n/\alpha)$. Such layout is described by the recursive procedure of Figure 6. The initial value for i, j and ℓ is 1 and ℓ is a global variable.

Figure 6. Storing the image into the leaves of the tree.

```
STORE(I, μ, i, j)
    if μ > 1
        STORE(I, μ/2, i, j)
        STORE(I, μ/2, i + μ/2, j)
        STORE(I, μ/2, i, j + μ/2)
        STORE(I, μ/2, i + μ/2, j + μ/2)
    else store A_{i,j} into leaf ℓ; ℓ = ℓ + 1
```

In parallel for each area, each leaf processor applies the sequential parsing algorithm so that in $\mathrm{O}(\alpha \log M)$ time each area is parsed into rectangles, some of which are monochromatic. Again, before encoding we wish to compute larger monochromatic rectangles. 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 $\mathrm{O}(n/(2^{2k} \log n))$ 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 6 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 $A_{(i-1)2^{k-1}+1,j}$, $A_{(i-1)2^{k-1}+2,j}$, $\cdots$ $A_{i2^{k-1},j}$, 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 $A_{i2^{k-1}+1,j}$, $A_{i2^{k-1}+2,j}$, $\cdots$ $A_{(i+1)2^{k-1},j}$ 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 $A_{i,(j-1)2^{k-1}+1}$, $A_{i,(j-1)2^{k-1}+2}$, $\cdots$ $A_{i,j2^{k-1}}$, 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 $A_{i,j2^{k-1}+1}$, $A_{i,j2^{k-1}+2}$, $\cdots$ $A_{i,(j+1)2^{k-1}}$ 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 [14], 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. 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. As previously mentioned, some areas could be entirely covered by a monochromatic match and 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. The interested reader can see how the decoding phase is implemented with the same complexity in [8].

7. Conclusions

In this paper, we presented a survey on parallel and distributed algorithms for Lempel–Ziv data compression. This field has developed in the last twenty years from a theoretical approach concerning parallel time complexity with no memory constraints to the practical goal of designing distributed algorithms with bounded memory and low communication cost. An extension to image compression has also been realized. As future work, we would like to implement distributed algorithms for Lempel–Ziv text compression and decompression on a parallel machine. As far as image compression is concerned, we wish to experiment with more processors on a graphical processing unit.

References

1. Lempel, A.; Ziv, J. On the Complexity of Finite Sequences. *IEEE Trans. Inf. Theory* **1976**, *22*, 75–81.
2. Lempel, A.; Ziv, J. A Universal Algorithm for Sequential Data Compression. *IEEE Trans. Inf. Theory* **1977**, *23*, 337–343.
3. Ziv, J.; Lempel, A. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Trans. Inf. Theory* **1978**, *24*, 530–536.
4. Crochemore, M.; Rytter, W. Efficient Parallel Algorithms to Test Square-freeness and Factorize Strings. *Inf. Proc. Lett.* **1991**, *38*, 57–60.
5. De Agostino, S. Parallelism and Dictionary-Based Data Compression. *Inf. Sci.* **2001**, *135*, 43–56.
6. De Agostino, S. P-complete Problems in Data Compression. *Theor. Comput. Sci.* **1994**, *127*, 181–186.

7. De Agostino, S.; Silvestri, R. Bounded Size Dictionary Compression: SC^k-Completeness and NC Algorithms. *Inf. Comput.* **2003**, *180*, 101–112.
8. Cinque, L.; De Agostino, S.; Lombardi, L. Scalability and Communication in Parallel Low-Complexity Lossless Compression. *Math. Comput. Sci.* **2010**, *3*, 391–406.
9. De Agostino, S. Almost Work-Optimal PRAM EREW Decoders of LZ-Compressed Text. *Parallel Proc. Lett.* **2004**, *14*, 351–359.
10. Belinskaya, D.; De Agostino, S.; Storer, J.A. Near Optimal Compression with respect to a Static Dictionary on a Practical Massively Parallel Architecture. *Proceedings IEEE Data Compression Conference 1995*; IEEE Computer Society: Washington, DC, USA, 1995; pp. 172–181.
11. Klein, S.T.; Wiseman, Y. Parallel Lempel-Ziv Coding. *Discrete Appl. Math.* **2005**, *146*, 180–191.
12. De Agostino, S. LZW *versus* Sliding window Compression on a Distributed System: Robustness and Communication. Unpublished work, 2011.
13. Storer, J.A. Lossless Image Compression using Generalized LZ1-Type Methods. *Proceedings IEEE Data Compression Conference 1996*; IEEE Computer Society: Washington, DC, USA, 1996; pp. 290–299.
14. Storer, J.A.; Helfgott, H. Lossless Image Compression by Block Matching. *Comput. J.* **1997**, *40*, 137–145.
15. Storer, J.A.; Szimansky, T.G. Data Compression via Textual Substitution. *J. ACM* **1982**, *24*, 928–951.
16. Rodeh, M.; Pratt, V.R.; Even, S. Linear Algorithms for Compression via String Matching. *J. ACM* **1980**, *28*, 16–24.
17. Mc Creight, E.M. A Space-Economical Suffix Tree Construction Algorithm. *J. ACM* **1976**, *23*, 262–272.
18. Welch, T.A. A Technique for High-Performance Data Compression. *IEEE Comput.* **1984**, *17*, 8–19.
19. De Agostino, S.; Storer, J.A. On-Line *versus* Off-line Computation for Dynamic Text Compression. *Inf. Proc. Lett.* **1996**, *59*, 169–174.
20. De Agostino, S.; Silvestri, R. A Worst Case Analisys of the LZ2 Compression Algorithm. *Inf. Comput.* **1997**, *139*, 258–268.
21. Storer, J.A. Data Compression: Methods and Theory. *Comput. Sci. Press* **1988**, 413.
22. Fiala, E.R.; Green, D.H. Data Compression with Finite Windows. *Commun. ACM* **1988**, *32*, 490–505.
23. Waterworth, J.R. Data Compression System. *U.S. Patent 4,701,745*, 1987.
24. Brent, R.P. A Linear Algorithm for Data Compression. *Aust. Comput. J.* **1987**, *19*, 64–68.
25. Whiting, D.A.; George, G.A.; Ivey, G.E. Data Compression Apparatus and Method. *U.S. Patent 5,016,009*, 1991.
26. Gailly, J.; Adler, M. The Gzip homepage. Available online: http://www.gzip.org (accessed on 6 September 2011).
27. Hartman, A.; Rodeh, M. Optimal Parsing of Strings. In *Comb. Algorithms Words*; Apostolico, A., Galil, Z., Eds.; Springer: Berlin, Germany, 1985; pp. 155–167.
28. Crochemore, M.; Rytter, W. *Jewels of Stringology*; World Scitific Publishing Co.: Singapore, 2003.

29. De Agostino, S. Bounded Size Dictionary Compression: Relaxing the LRU Deletion Heuristic. *Int. J. Found. Comput. Sci.* **2006**, *17*, 1273–1280.
30. Farach M.; Muthikrishnan, S. Optimal Parallel Dictionary Matching and Compression. *Proc. SPAA* **1995**, 244–253.
31. De Agostino, S. Work-Optimal Parallel Decoders for LZ2 Data Compression. *Proceedings IEEE Data Compression Conference 2000*; IEEE Computer Society: Washington, DC, USA, 2000; pp. 393–399.
32. Bell, T.C.; Cleary, J.G.; Witten, I.H. Text Compression. *Prentice Hall* **1990**.
33. Rissanen, J.; Langdon, G.G. Universal Modeling and Coding. *IEEE Trans. Inf. Theory* **1981**, *27*, 12–23.
34. Rissanen, J. Generalized Kraft Inequality and Arithmetic Coding. *IBM J. Res. Dev.* **1976**, *20*, 198–203.
35. Howard, P.G.; Kossentini, F.; Martinis, B.; Forchammer, S.; Rucklidge, W.J.; Ono, F. The Emerging JBIG2 Standard. *IEEE Trans. Circuits Syst. Video Technol.* **1998**, *8*, 838–848.
36. Wu, X.; Memon, N.D. Context-Based, Adaptive, Lossless Image Coding. *IEEE Trans. Commun.* **1997**, *45*, 437–444.
37. Weimberger, M.J.; Seroussi, G.; Sapiro, G.; LOCO-I: A Low Complexity, Context Based, Lossless Image Compression Algorithm. *Proceedings IEEE Data Compression Conference 1996*, IEEE Computer Society: Washington, DC, USA, 1996; pp. 140–149.
38. Golomb, S.W. Run-Length Encodings. *IEEE Trans. Inf. Theory* **1996**, *12*, 399–401.
39. Rice, R.F. Some Practical Universal Noiseless Coding Technique—part I; Technical Report JPL-79-22; Jet Propulsion Laboratory: Pasadena, CA, USA, 1979.
40. Cinque, L.; De Agostino, S.; Liberati, F.; Westgeest, B. A Simple Lossless Compression Heuristic for Grey Scale Images. *Int. J. Found. Comput. Sci.* **2005**, *16*, 1111–1119.
41. Cinque, L.; De Agostino, S.; Lombardi, L. Binary Image Compression via Monochromatic Pattern Substitution: Effectiveness and Scalability. *Proc. Prague Stringol. Conf.* **2010**, 103–115.