LZW Data Compression on Large Scale and Extreme Distributed Systems

Sergio De Agostino

Computer Science Department, Sapienza University, 00198 Rome, Italy

Abstract. Results on the parallel complexity of Lempel-Ziv data compression suggest that the sliding window method is more suitable than the LZW technique on shared memory parallel machines. When instead we address the practical goal of designing distributed algorithms with low communication cost, sliding window compression does not seem to guarantee robustness if we scale up the system. The possibility of implementing scalable heuristics is instead offered by LZW compression. In this paper we present two implementations of the LZW technique on a large scale and an extreme distributed system, respectively. They are both derived from a parallel approximation scheme of a bounded memory version of the sequential algorithm.

Keywords: compression, factorization, distributed system, scalability

1 Introduction

Lempel-Ziv compression [18], [19], [22] is based on string factorization. Two different factorization processes exist with no memory constraints. With the first one (LZ1) [19], 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 (sliding window compression). With the second one (LZ2 or LZW) [22], 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 sliding window compression has efficient parallel algorithms [6], [11], [12], [3], LZW compression (a practical implementation of the LZ2 method [21]) is hard to parallelize [5]. This difference is maintained when the most effective bounded memory versions of Lempel-Ziv compression are considered [15], [2]. There are several heuristics for limiting the work-space of the LZW compression procedure in the literature. The most effective is the "least recently used" strategy (LRU). Hardness results inside Steve Cook's class (SC) have been proved for this approach [15], implying the likeliness of the non-inclusion of the LZW-LRU compression method in Nick Pippenger's class (NC). Completeness results in SC have also been obtained for a relaxed version of the LRU strategy (RLRU) [15]. RLRU was shown to be as effective as LRU in [8], [9]. Therefore, RLRU is the most efficient among the bounded memory versions of LZW compression. A simpler heuristic which is still effective is the RESTART strategy. Differently from LRU and RLRU, LZW-RESTART is parallelizable [15]. Moreover, parallel decompression is possible (this is true also for the unbounded memory version) [6], [7], [11], [12].

When we address the practical goal of designing distributed algorithms with low communication cost sliding window compression does not seem to guarantee robustness when we scale up the system [10], [11], [12], [2]. The possibility of implementing scalable heuristics is instead offered by LZW-RESTART compression [11], [12], [13]. Traditionally, the scale of a system is considered large when the number of nodes has

Sergio De Agostino: LZW Data Compression on Large Scale and Extreme Distributed Systems, pp. 18–27.
Proceedings of PSC 2012, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05095-8 © Czech Technical University in Prague, Czech Republic

the order of magnitude of a thousand. Modern distributed systems may nowadays consist of hundreds of thousands of nodes, pushing scalability well beyond traditional scenarios (extreme distributed systems). In this paper we present two implementations of the LZW technique on a large scale and an extreme distributed system, respectively. They are both derived from a parallel approximation scheme of the bounded memory version of the sequential algorithm presented in [13]. The approach for extreme distributed systems could be applied to arbitrarily smaller scale systems as well, but the alternative implementation we propose is simpler.

In Section 2 we describe Lempel-Ziv data compression while the bounded memory versions are given in Section 3. Section 4 briefly describes past work on the study of the parallel complexity of Lempel-Ziv methods since it is somehow consistent with the practical results on the distributed implementation of LZW compression shown in Section 5. Conclusion and future work are given in Section 6.

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. 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 the factor 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 a 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 $f_1 f_2 \cdots f_{i-1}$ [20]. 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.

2.2 LZ2 Compression

The LZ2 factorization of a string S is $S = f_1 f_2 \cdots f_i \cdots f_k$ where the factor 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 [21]. 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 one 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 [16] and the greedy algorithm approximates with an $O(n^{\frac{1}{4}})$ multiplicative factor the optimal solution [14].

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 sliding window compression this can be simply obtained by bounding the match and window lengths (therefore, the left end of the window slides as well), for the LZW compression the dictionary elements are removed by using a deletion heuristic.

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 the FREEZE 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 a 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. Usually, the dictionary performs well in a static way on a block long enough to learn another dictionary of the same size. This is what is done by the SWAP heuristic. When the other dictionary is filled, they swap their roles on the successive block.

The best deletion heuristic is the LRU (last recently used) strategy. The LRU deletion heuristic removes elements from the dictionay 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. In [15] a relaxed version (RLRU) was introduced. 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 [8], [9]. 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. SWAP is the best heuristic among the "discrete" ones.

3.2 Compression with Finite Windows

As mentioned at the beginning of this section, bounded size dictionary compression can also be obtained by sliding a fixed length window and by bounding the match length. 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.

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, RESTART or SWAP 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 [17], [4]. 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. However, greedy factorizations are very close to optimal in practice even if they approximate the optimal solution with a multiplicative factor equal to the maximum match length in the worst case.

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; $j := k'$; $i := h(k')$

Figure 1. The semi-greedy factorization procedure

4 Lempel-Ziv Compression on a Parallel System

LZSS (or LZ1) compression can be efficiently parallelized on a PRAM EREW [6], [2], [3], 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 [5] and, therefore, hard to parallelize. Decompression, instead, is parallelizable for both methods [7]. The asymmetry of the pair encoder/decoder between LZ1 and LZ2 follows from the fact that the hardness results of the LZ2/LZW encoder depend on the factorization process rather than on the coding itself.

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 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. The SC^k-hardness and SC^k-completeness of LZ2 compression using, respectively, the LRU and RLRU deletion heuristics and a dictionary of polylogarithmic size show that it is unlikely to have a parallel complexity involving reasonable multiplicative constants [15]. In conclusion, the only practical LZW compression algorithm for a shared memory parallel system is the one using the FREEZE, RESTART or SWAP deletion heuristics. Unfortunately, the SWAP heuristic does not seem to have a parallel decoder. Since the FREEZE heuristic is not very effective in terms of compression, RESTART is a good candidate for an efficient parallel implementation of the pair encoder/decoder even on a distributed system. We will see these arguments more in detail in the next section.

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. Such array of processors could be a set of neighbors linked directly to a central node (from which they receive blocks of the input) to form a so called *star* network (a rooted tree of height 1). In an *extended* star each node adjacent to the central one has a set of leaf neighbors (a rooted tree of height 2). Such extension is useful in practice when we scale up the system.

For every integer k greater than 1 there is 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 [2]. 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 $O(\ell)$ time, $O(n/\ell)$ processors distributed algorithm. This could also be done with the LRU or SWAP deletion heuristic. However, with the RESTART deletion heuristic scalable compression and decompression algorithms are possible on a tree architecture. The parallel encoder, after a dictionary is filled for each block of length ℓ, produces a factorization of S with a cost approximating the cost of the optimal factorization within the multiplicative

factor $(k+1)/k$ in $O(km)$ time with $O(n/km)$ processors [13]. These algorithms provide approximation schemes for the corresponding factorization problems since the approximation factors converge to 1 when km and kw converge to ℓ and to n, respectively. In the following subsections, we first discuss sliding window compression and then propose two improved new versions of the LZW distributed algorithm suitable on large scale and extreme distributed systems.

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 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.

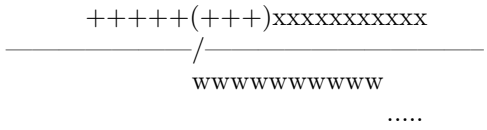

Figure 2. The making of the surplus factors

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. Since the compression tools of the Zip family use a window size of at least 32 K, 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. Therefore, the approximation scheme is suitable only for a small scale system unless the file size is very large.

5.2 LZW Compression on a Distributed System

LZW compression was originally presented with a dictionary of size 2^{12}, clearing out the dictionary as soon as it is filled up [21]. The Unix command "compress" employs a dictionary of size 2^{16} and works with the RESTART deletion heuristic. The block length needed to fill up a dictionary of this size is approximately 300 K.

As previously mentioned, the SWAP heuristic is the best deletion heuristic among the discrete ones. After a dictionary is filled up on a block of 300 K, the SWAP heuristic shows that we can use it efficiently on a successive block of about the same dimension where a second dictionary is learned. A distributed compression algorithm employing the SWAP heuristic learns a different dictionary on every block of 300 K of a partitioned string (the first block is compressed while the dictionary is learned). For the other blocks, block i is compressed statically in a second phase using the dictionary

learned during the first phase on block $i-1$. But, unfortunately, the decoder is not parallelizable since the dictionary to decompress block i is not available until the previous blocks have been decompressed. On the other hand, with RESTART we can work on a block of 600 K where the second half of it it is compressed statically. We wish to speed up this second phase though, since LZW compression must be kept more efficient than sliding window compression. In fact, it is well-known that sliding window compression is more effective but slower. If both methods are applied to a block of 300 K, but LZW has a second static phase to execute on a block of about the same length it would no longer have the advantage of being faster. We show how to speed up this second phase on a very simple tree architecture as the extended star in time $O(km)$ with $O(n/km)$ processors.

During the input phase, the central node broadcasts a block of length 600 K to each adjacent processor. Then, for each block the corresponding processor broadcasts to the adjacent leaves a sub-block of length $m(k+2)$ in the suffix of length 300 K, except for the first one and the last one which are $m(k+1)$ long. Each sub-block overlaps on m characters with the adjacent sub-block to the left and to the right, respectively (obviously, the first one overlaps only to the right and the last one only to the left). Every processor stores a dictionary initially set to comprise only the alphabet characters.

The first phase of the computation is executed by processors adjacent to the central node. The prefix of length 300 K of each block is compressed while learning the dictionary. At each step of the LZW factorization process, each of these processors sends the current factor to the adjacent leaves. They all adds such factor to their own dictionary. After compressing the prefix of length 300 K of each block, all the leaves have a dictionary stored which has been learned by their parents during such compression phase.

Let us call a *boundary match* a factor covering positions of two adjacent sub-blocks stored by leaf processors. Then, the leaf processors execute the following algorithm to compress the suffix of length 300 K of each block:

- for each block, every corresponding leaf processor but the one associated with the last sub-block computes the boundary match between its sub-block and the next one ending furthest to the right, if any;

- each leaf processor computes the optimal factorization from the beginning of its sub-block to the beginning of the boundary match on the right boundary of its sub-block (or the end of its sub-block if there is no boundary match).

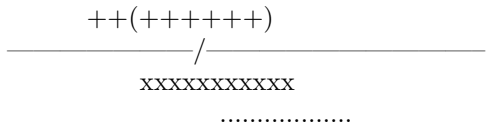

Figure 3. The making of a surplus factor

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 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 [1], it is shown experimentally that for $k = 10$ the compression ratio achieved by such factorizarion is about the same as the sequential one. Results were presented for static prefix dictionary compression but they are valid for dynamic compression using the LZW technique with the RESTART deletion heuristic. In fact, experiments were proposed compressing similar files in a collection using a dictionary learned from one of them. This is true even if the second step is greedy, since greedy is very close to optimal in practice. Moreover, with the greedy approach it is enough to use a simple trie data structure for the dictionary rather than the modified suffix tree data structure of [17] needed to implement the semi-greedy factorization in real time. Therefore, after computing the boundary matches the second part of the parallel approximation scheme can be substituted by the following procedure:

– each leaf processor computes the static greedy factorization from the end of the boundary match on the left boundary of its sub-block to the beginning of the boundary match on the right boundary.

Considering that typically the average match length is 10, one processor can compress down to 100 bytes independently. Then compressing 300 K involves a number of processors up to 3000 for each block. It follows that with a file size of several megabytes or more, the system scale has a greater order of magnitude than the standard large scale parameter making the implementation suitable for an extreme distributed system. We wish to point out that the computation of the boundary matches is very relevant for the compression effectiveness when an extreme distributed system is employed since the sub-block length becomes much less than 1 K.

With standard large scale systems the sub-block length is several kilobytes with just a few megabytes to compress and the approach using boundary matches is too conservative for the static phase. In fact, a partition of the second half of the block does not effect on the compression effectiveness unless the sub-blocks are very small since the process is static. In conclusion, we can propose a further simplification of the algorithm for standard small, medium and large scale distributed systems.

Let $p_0 \cdots p_n$ be the processors of a distributed system with an extended star topology. p_0 is the central node of the extended star network and $p_1 \cdots p_m$ are its neighbors. For $1 \leq i \leq m$ and $t = (n - m)/m$ let the processors $p_{m+(i-1)t+1} \cdots p_{m+it}$ be the neighbors of processor i.

$B_1 \cdots B_m$ is the sequence of blocks of length 600 K partitioning the input file. Denote with B_i^1 and B_i^2 the two halves of B_i for $1 \leq i \leq m$. Divide B_i^2 into t sub-blocks of equal length.

The input phase of this simpler algorithm distributes for each block the first half and the sub-blocks of the second half in the following way:

– broadcast B_i^1 to processor p_i for $1 \leq i \leq m$

– broadcast the j-th sub-block of B_i^2 to processor $p_{m+(i-1)t+j}$ for $1 \leq i \leq m$ and $1 \leq j \leq t$

Then, the computational phase is:

in parallel for $1 \leq i \leq m$

- processor p_i applies LZW compression to its block, sending the current factor to its neigbors at each step of the factorization
- the neighbors of processor p_i compress their blocks statically using the dictionary received from p_i with a greedy factorization

5.3 Decompression

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 LZW compressor implemented with one of the two procedures described in the previous section, a second special mark indicates for each block the end of the coding of a sub-block. The coding of the first half of each block is stored in one of the neighbors of the central node while the coding of the sub-blocks are stored into the corresponding leaves. The first half of each block is decoded by one processor to learn the corresponding dictionary. Each decoded factor is sent to the corresponding leaves during the process, so that the leaves can rebuild the dictionary themselves. Then, the dictionary is used by the leaves to decode the sub-blocks of the second half.

6 Conclusion

We presented an approach to the parallel implementation of LZW data compression which is suitable for small and large scale distributed systems. Some blocks are compressed independently providing information for a second phase where the remaining portions of the input string are encoded in parallel with a higher granularity. In order to push scalability beyond what is traditionally considered a large scale system a more involved approach distributes overlapping sub-blocks of these remaining portions to compute boundary matches. These boundary matches are relevant to maintain the compression effectiveness on a so-called extreme distributed system. We wish to implement these ideas on real systems with the appropriate architecture to experiment how the communication cost effects on the speed-up. If we have a relatively small scale system available, the approach with no bounadary matches can be used. Moreover, if the system has an architecture with a simple star topology rather than an extended one we could still experiment on a file with size between 500 K and one megabyte. During the input phase, the central node brodcasts the sub-blocks of the second half of the file to the neighbors. Then, it applies LZW compression to the first half providing the dictionary to the other nodes for the compression of the second half. The parallel running time of this implementation could be compared with the sequential time of the sliding compression method applied to each of the two halves of the file (the higher one would be considered). In this way, it can be seen how the running times of the two parallel implementations relate to each other.

References

1. D. Belinskaya, S. D. Agostino, and J. A. Storer: *Near optimal compression with respect to a static dictionary on a practical massively parallel architecture*, in Proceedings IEEE Data Compression Conference, 1996, pp. 172–181.
2. L. Cinque, S. DeAgostino, and L. Lombardi: *Scalability and communication in parallel low-complexity lossless compression.* Mathematics in Computer Science, 3 2010, pp. 391–406.
3. M. Crochemore and W. Rytter: *Efficient parallel algorithms to test square-freeness and factorize strings.* Information Processing Letters, 38 1991, pp. 57–60.
4. M. Crochemore and W. Rytter: *Jewels of Stringology*, World Scientific, 2003.
5. S. DeAgostino: *P-complete problems in data compression.* Theoretical Computer Science, 127 1994, pp. 181–186.
6. S. DeAgostino: *Parallelism and dictionary-based data compression.* Information Sciences, 135 2001, pp. 43–56.
7. S. DeAgostino: *Almost work-optimal PRAM EREW decoders of LZ-compressed text.* Parallel Processing Letters, 14 2004, pp. 351–359.
8. S. DeAgostino: *Bounded size dictionary compression: Relaxing the LRU deletion heuristic*, in Proceedings Prague Stringology Conference, 2005, pp. 135–142.
9. S. DeAgostino: *Bounded size dictionary compression: Relaxing the LRU deletion heuristic.* International Journal of Foundations of Computer Science, 17 2006, pp. 1273–1280.
10. S. DeAgostino: *Parallel implementations of dictionary text compression without communication*, 2009.
11. S. DeAgostino: *Lempel-Ziv data compression on parallel and distributed systems*, in Proceedings Data Compression, Communications and Processing Conference, 2011, pp. 193–202.
12. S. DeAgostino: *Lempel-Ziv data compression on parallel and distributed systems.* Algorithms, 4 2011, pp. 183–199.
13. S. DeAgostino: *LZW versus sliding window compression on a distributed system: Robustness and communication*, in Proceedings INFOCOMP, 2011, pp. 125–130.
14. S. DeAgostino and R. Silvestri: *A worst case analisys of the LZ2 compression algorithm.* Information and Computation, 139 1997, pp. 258–268.
15. S. DeAgostino and R. Silvestri: *Bounded size dictionary compression:* SC^k*-completeness and* nc *algorithms.* Information and Computation, 180 2003, pp. 101–112.
16. S. DeAgostino and J. A. Storer: *On-line versus off-line computation for dynamic text compression.* Information Processing Letters, 59 1996, pp. 169–174.
17. A. Hartman and M. Rodeh: *Optimal parsing of strings*, 1985.
18. A. Lempel and J. Ziv: *On the complexity of finite sequences.* IEEE Transactions on Information Theory, 22 1976, pp. 75–81.
19. A. Lempel and J. Ziv: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, 23 1977, pp. 337–343.
20. J. A. Storer and T. G. Szymanski: *Data compression via textual substitution.* Journal of ACM, 29 1982, pp. 928–951.
21. T. A. Welch: *A technique for high-performance data compression.* IEEE Computer, 17 1984, pp. 8–19.
22. J. Ziv and A. Lempel: *Compression of individual sequences via variable-rate coding.* IEEE Transactions on Information Theory, 24 1978, pp. 530–536.