Compressing Large Size Files on the Web in MapReduce

Sergio De Agostino
Computer Science Department
Sapienza University
Rome, Italy
Email: deagostino@di.uniroma1.it

***Abstract*—Lempel-Ziv (LZ) techniques are the most widely used for lossless file compression. LZ compression basicly comprises two methods, called LZ1 and LZ2. The LZ1 method is the one employed by the family of Zip compressors, while the LZW compressor implements the LZ2 method, which is slightly less effective but twice faster. When the file size is large, both methods can be implemented on a distributed system guaranteeing linear speed-up, scalability and robustness. With Web computing, the MapReduce model of distributed processing is emerging as the most widely used. In this framework, we present and make a comparative analysis of different implementations of LZ compression. An alternative to standard versions of the Lempel-Ziv method is proposed as the most efficient one for large size files compression.**

Keywords-web computing; mapreduce framework; lossless compression; string factorization

I. INTRODUCTION

Lempel-Ziv (LZ) techniques are the most widely used for lossless file compression. LZ compression [1], [2], [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 sliding window compression has efficient parallel algorithms [4], [5], [6], [7], LZ2 compression is hard to parallelize [8] and less effective in terms of compression. On the other hand, LZ2 is more efficient computationally than sliding window compression from a sequential point of view. This difference is maintained when the most effective bounded memory versions of Lempel-Ziv compression are considered [6], [9]. While the bounded memory version of LZ1 compression is quite straightforward, there are several heuristics for limiting the work-space of the LZ2 compression procedure in the literature. The "least recently used" strategy (LRU) is the most effective one. Hardness results inside Steve Cook's class (SC) have been proved for this approach [9], implying the likeliness of the non-inclusion of the LZ2-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) [9]. RLRU was shown to be as effective as LRU in [10] and, consequently, it is the most efficient one among the Lempel-Ziv techniques.

Bounding memory is very relevant with distributed processing and it is an important requirement of the MapReduce model of computation for Web computing. A formalization of this model was provided in [11], where further constraints are formulated for the number of processors, the number of iterations and the running time. However, such constraints are a necessary but not sufficient condition to guarantee a robust linear speed-up. In fact, interprocessor communication is allowed during the computational phase and experiments are needed to verify an actual speed-up. Distributed algorithms for the LZ1 and LZ2 methods approximating in practice their compression effectiveness have been realized in [6], [12], [13], where the stronger requirement of no interprocessor communication during the computational phase is satisfied. In fact, the approach to a distributed implementation in this context consists of applying the sequential procedure to blocks of data independently.

In Sections 2 and 3, we describe the Lempel-Ziv compression techniques and their bounded memory versions respectively. Section 4 sketches past work on the study of the parallel complexity of Lempel-Ziv methods leading to the idea of relaxing the least recently used strategy. In Section 5, we present the MapReduce model of computation and introduce further constraints for a robust approach to a distributed implementation of LZ compression on the Web. Section 6 makes a comparative analysis of different implementations of LZ compression in this framework and proposes an alternative to the standard versions as the most efficient one for large size files compression. Conclusions and future work are given in Section 7.

II. 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.

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

 ISBN: 978-1-61208-280-6

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 $f_1 f_2 \cdots f_{i-1}$ [14]. 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.

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 [15]. 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.

III. 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. For LZSS (or LZ1) compression this can be simply obtained by sliding a fixed length window and by bounding the match length. 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. 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, SWAP, LRU [16] and RLRU [9].

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). 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 [9] 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 [10]. 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.

IV. LZ Compression on a Parallel System

LZSS (or LZ1) compression can be efficiently parallelized on a PRAM EREW [4], [5], 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 [8] and, therefore, hard to parallelize. Decompression, instead, is parallelizable for both methods [17]. 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 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

 ISBN: 978-1-61208-280-6

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 [9]. 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 on a shared memory parallel system and even on a system with distributed memory. However, in the context of distributed processing of massive data with no interprocessor communication the LZW-RLRU technique turns out to be the most efficient one. We will see these arguments more in detail in the next two sections.

V. The MapReduce Model of Computation

The MapReduce programming paradigm is a sequence $P = \mu_1\rho_1 \cdots \mu_R\rho_R$ where μ_i is a mapper and ρ_i is a reducer for $1 \leq i \leq R$. First, we describe such paradigm and then discuss how to implement it on a distributed system. Distributed systems have two types of complexity, the interprocessor communication and the input-output mechanism. The input/output issue is inherent to any parallel algorithm and has standard solutions. In fact, in [11] the sequence P does not include the I/O phases and the input to μ_1 is a multiset U_0 where each element is a $(key, value)$ pair. The input to each mapper μ_i is a multiset U_{i-1} output by the reducer ρ_{i-1} for $1 < i \leq R$. Mapper μ_i is run on each pair (k, v) in U_{i-1}, mapping (k, v) to a set of new $(key, value)$ pairs. The input to reducer ρ_i is U'_i, the union of the sets output by μ_i. For each key k, ρ_i reduces the subset of pairs of U'_i with the key component equal to k to a new set of pairs with key component still equal to k. U_i is the union of these new sets.

In a distributed system implementation, a key is associated with a processor (a node in the Web). All the pairs with a given key are processed by the same node but more keys can be associated to it in order to lower the scale of the system involved. Mappers are in charge of the data distribution since they can generate new key values. On the other hand, reducers just process the data stored in the distributed memory since they output for a set of pairs with a given key another set of pairs with the same given key.

To add the I/O phases to P, we extend the sequence to $\mu_0\rho_0\mu_1\rho_1 \cdots \mu_R\rho_R\mu_{R+1}\rho_{R+1}$, where (λ, x) is the unique $(key, value)$ pair input to μ_0 with λ empty key and x input data. μ_0 distributes the data generating the multiset U_0 while ρ_0 is the identity function. Finally, μ_{R+1} maps U_R to a multiset where all the pair elements have the same key λ and ρ_{R+1} reduces such multiset to the pair (λ, y) where y is the output data.

The following complexity requirements are stated as necessary for a practical interest in [11]:

- R is polylogarithmic in the input size n;
- the number of processors (or nodes in the Web) involved is $O(n^{1-\epsilon})$ with $0 < \epsilon < 1$;
- the amount of memory available for each node is $O(n^{1-\epsilon})$;
- the running time of mappers and reducers is polynomial in n.

As mentioned in the introduction, such requirements are necessary but not sufficient to guarantee a speed-up of the computation. Obviously, the total running time of mappers and reducers cannot be higher than the sequential one and this is trivially implicit in what is stated in [11]. The non-trivial bottleneck is the communication cost of the computational phase after the distribution of the original input data among the processors and before the output of the final result. This is obviously algorithm-dependent and needs to be checked experimentally since R can be polylogarithmic in the input size. The only way to guarantee with absolute robustness a speed-up with the increasing of the number of nodes is to design distributed algorithms implementable in MapReduce with $R = 1$. Moreover, if we want the speed-up to be linear then the total running time of mappers and reducers must be $O(t(n)/n^{1-\epsilon})$ where $t(n)$ is the sequential time. These stronger requirements are satisfied by the distributed implementations of the several versions of LZ compression discussed in the next section, except for one of them, which requires $R = 2$.

VI. LZ Compression on the Web in MapReduce

We can factorize blocks of length ℓ of an input string using any of the bounded memory compression techniques with an $O(\ell)$ time, $O(n/\ell)$ processors distributed algorithm. The algorithm is suitable for a small scale system but due to its adaptiveness it works on a large scale parallel system only when the file size is large.

With the sliding window method, ℓ is equal to kw where k is a positive integer and w is the window length [6], [12], [13]. 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 32K. 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 guarantees robustness on realistic data. The window length is

 ISBN: 978-1-61208-280-6

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 32K. It follows that the block length in our parallel implementation should be about 300K and the file size should be about one third of the number of processors in megabytes.

In the MapReduce framework, we implement the distributed procedure above with a sequence $\mu_0\rho_0\mu_1\rho_1\mu_2\rho_2$ where $\mu_0\rho_0$ and $\mu_2\rho_2$ are the input and output phases, respectively. Let $X = X_1 \cdots X_m$ be the input string where X_i is a substring that has the same length $\ell \geq 300K$ for $1 \leq i \leq m$. The complexity requirements of the MapReduce model will be satisfied by the fact that ℓ is allowed to be strictly greater than 300K. The input to μ_0 is the pair $(0, X)$ mapping this element to the set S of pairs $(1, X_1) \cdots (m, X_m)$ and the reducer ρ_0 sets U_0 to S as input to μ_1. U_0 is mapped to itself by μ_1 and ρ_1 reduces (i, X_i) to (i, Y_i) where Y_i is the LZSS coding of X_i for $1 \leq i \leq m$. Finally, μ_2 maps each element (i, Y_i) of its input $U_1 = \{(1, Y_1) \cdots (m, Y_m)\}$ to $(0, Y_i)$ and ρ_2 outputs $(0, Y)$ where $Y = Y_1 \cdots Y_m$. Obviously, the stronger requirements for a linear speed-up, stated in the previous section, are satisfied by this program.

As far as LZW compression is concerned, it was originally presented with a dictionary of size 2^{12}, clearing out the dictionary as soon as it is filled up [15]. 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 300K. 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 300K, 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 300K 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 600K where the second half of 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 300K and 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 showed how to speed up in a scalable way this second phase on a very simple tree architecture as the extended star network in [12], [18]. The idea is to factorize small sub-blocks of at least 100 bytes of the second half in parallel. This is possible with no relevant loss of compression effectiveness since the dictionary has already been learned and the factorization is static.

In the MapReduce framework, the program sequence is $\mu_0\rho_0\mu_1\rho_1\mu_2\rho_2\mu_3\rho_3$ where $\mu_0\rho_0$ and $\mu_3\rho_3$ are the input and output phases, respectively. Let $X = X_1Y_1 \cdots X_mY_m$ be the input string where X_i and Y_i are substrings having the same length $\ell \geq 300K$ for $1 \leq i \leq m$ and $Y_i = B_{i,1} \cdots B_{i,r}$ such that $B_{i,j}$ is a substring that has the same length $\ell' \geq 100$ for $1 \leq j \leq r$. The complexity requirements of the MapReduce model will be satisfied by the fact that ℓ is allowed to be strictly greater than 300K and ℓ' strictly greater than 100 bytes. Keys are pairs of positive integers. The input to μ_0 is the pair $((0, 0), X)$, which is mapped to the set S of pairs $((0, 1), X_1)$, $((1, 1), B_{1,1})$, $\cdots$, $((1, r), B_{1,r})$, $\cdots$, $((0, m), X_m))$, $((m, 1), B_{m,1})$, $\cdots$, $((m, r), B_{m,r})$ and the reducer ρ_0 sets U_0 to S as input to μ_1. U_0 is mapped to itself by μ_1. ρ_1 reduces $((0, i), X_i)$ to a set of two $(key, value)$ pairs, that is, $\{((0, i), Z_i), ((0, i), D_i)\}$, where Z_i and D_i are respectively the LZW coding of X_i and the dictionary learned during the coding process. On the other hand, $((i, j), B_{i,j})$ are reduced to themselves by ρ_1 for $1 \leq i \leq m$ and $1 \leq j \leq r$. The second iteration step $\mu_2\rho_2$ works as the identity function when applied to $((0, i), Z_i)$. μ_2 works as the identity function when applied to $((i, j), B_{i,j})$ as well. Instead, $((0, i), D_i)$ is mapped by μ_2 to $((i, j), D_i)$ for $1 \leq j \leq r$. Then, ρ_2 reduces the set $\{((i, j), B_{i,j}), ((i, j), D_i)\}$ to $((i, j), Z_{i,j})$ where $Z_{i,j})$ is the coding produced by the second phase of LZW compression with the static dictionary D_i. Finally, μ_3 maps (i, Z_i) to $((0, 0), Z_i)$ and $((i, j), Z_{i,j})$ to $((0, 0), Z_{i,j})$. Then, ρ_3 outputs $((0, 0), Z)$ where $Z = Z_1Z_{1,1} \cdots Z_{1,r} \cdots Z_mZ_{m,1} \cdots Z_{m,r}$.

The communication cost during the computational phase of the MapReduce program above is determined by μ_2. The dictionary D_i is sent from the node associated with the key $(0, i)$ to the node associated with the key (i, j) in parallel for $1 \leq i \leq m$ and $1 \leq j \leq r$. Each factor f in D_i can be represented as pc where p is the pointer to the longest proper prefix of f (an element of D_i) and c is the last character of f. Since the standard sizes for the dictionary and the alphabet are respectively 2^{16} and 256, three bytes can represent a dictionary element. Conservatively, at least ten nanoseconds are spent to send a byte between nodes. Therefore, the communication cost to send a dictionary is at least $30(2^{16})$ nanoseconds, which is about two milliseconds. Considering the fact that 300K are compressed usually in about 30 milliseconds by a Zip compressor and in about 15 milliseconds by an LZW compressor, the communication cost is acceptable.

The approach described above is not robust when the data are highly disseminated [19]. However, when compressing large size files even on a large scale system the size of the blocks distributed among the nodes is larger than 600K.

 ISBN: 978-1-61208-280-6

In order to increase robustness when the data are highly disseminated, the most appropriate approach is to apply a procedure where no static phase is involved. Therefore, new dictionary elements should be learned at every step while bounding the dictionary size by means of a deletion heuristic. It is, then, reasonable to propose LZW-RLRU as the most suitable in this context since it is the most efficient one. The relaxed version (RLRU) of the LRU heuristic is:

> RLRUp: When the dictionary is not full, label the i^{th} element added to the dictionary with the integer $\lceil i \cdot p/k \rceil$, where k is the dictionary size minus the alphabet size and $p < k$ is the number of labels. When the dictionary is full, label the $i-th$ element with p if $\lceil i \cdot p/k \rceil = \lceil (i-1)p/k \rceil$. If $\lceil i \cdot p/k \rceil > \lceil (i-1)p/k \rceil$, decrease by 1 all the labels greater or equal to 2. Then, label the $i-th$ element with p. Finally, remove one of the elements represented by a leaf with the smallest label.

In other words, RLRU works with a partition of the dictionary in p classes, sorted somehow in a fashion according to the order of insertion of the elements in the dictionary, and an arbitrary element from the oldest class with removable elements is deleted when a new element is added. Each class is implemented with a stack. Therefore, the newest element in the class of least recently used elements is removed. Observe that if RLRU worked with only one class, after the dictionary is filled up the next element added would be immediately deleted. Therefore, RLRU would work like FREEZE. But for $p = 2$, RLRU is already more sophisticated than SWAP since it removes elements in a continuous way and its compression effectiveness compares to the original LRU. Therefore, LZW-RLRU2 is the most efficient approach to compress on the Web or any other distributed system when the size of the input file is very large. In the MapReduce framework, a program sequence $\mu_0\rho_0\mu_1\rho_1\mu_2\rho_2$ implements it as the one for the LZSS compressor explained at the beginning of this section.

Decompression in MapReduce is simmetrical. To decode the compressed files on a distributed system, it is enough to use a special mark occurring in the sequence of pointers where 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 using a second phase with a static dictionary, a second special mark indicates for each block the end of the coding of a sub-block. The input phase distributes the coding of the first half of each block and the coding of the sub-blocks of the second half. Then, two iterations as for the compression case decompress in MapReduce. The first one decodes the first half of each block and learns the corresponding dictionary. The second sends the dictionaries to the corresponding processors for the decoding of the sub-blocks of the second half.

VII. Conclusion

We showed how to implement Lempel-Ziv data compression in the MapReduce framework for Web computing. With large size files, the robustness of the approach is preserved with scalability since no interprocessor communication is required. It follows that a linear speed-up is guaranteed during the computational phase. With arbitrary size files, scaling up the system is necessary to preserve the efficiency of LZW compression with very low communication cost if the data are not highly disseminated. The MapReduce framework allows in theory a higher degree of communication than the one employed in the procedures presented in this paper. In [11], it has been shown how the PRAM model of computation can be simulated in MapReduce under specific constraints with the theoretical framework. These constraints are satisfied by several PRAM Lempel-Ziv compression and decompression algorithms designed in the past [5], which are suitable for arbitrary size highly disseminated files. As future work, it is worth investigating experimentally if any of these algorithms can be realized with MapReduce in practice on specific files.

References

[1] A. Lempel and J. Ziv, "On the Complexity of Finite Sequences," IEEE Transactions on Information Theory, vol. 22, 1976, pp. 75-81.

[2] A. Lempel and J. Ziv, "A Universal Algorithm for Sequential Data Compression," IEEE Transactions on Information Theory, vol. 23, 1977, pp. 337-343.

[3] J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-Rate Coding," IEEE Transactions on Information Theory, vol. 24, 1978, pp. 530-536.

[4] M. Crochemore and W. Rytter, "Efficient Parallel Algorithms to Test Square-freeness and Factorize Strings," Information Processing Letters, vol. 38, 1991, pp. 57-60.

[5] S. De Agostino, "Parallelism and Dictionary-Based Data Compression," Information Sciences, vol. 135, 2001, pp. 43-56.

[6] L. Cinque, S. De Agostino and L. Lombardi, "Scalability and Communication in Parallel Low-Complexity Lossless Compression," Mathematics in Computer Science, vol. 3, 2010, pp. 391-406.

[7] S. De Agostino, "Lempel-Ziv Data Compression on Parallel and Distributed Systems," Algorithms, vol. 4, 2011, pp. 183-199.

[8] S. De Agostino, "P-complete Problems in Data Compression," Theoretical Computer Science, vol. 127, 1994, pp. 181-186.

[9] S. De Agostino and R. Silvestri, "Bounded Size Dictionary Compression: SC^k-Completeness and NC Algorithms," Information and Computation, vol. 180, 2003, pp. 101-112.

 ISBN: 978-1-61208-280-6

[10] S. De Agostino, "Bounded Size Dictionary Compression: Relaxing the LRU Deletion Heuristic," International Journal of Foundations of Computer Science, vol. 17, 2006, pp. 1273-1280.

[11] H. J. Karloff, S. Suri and S. Vassilvitskii, "A Model of Computation for MapReduce," Proc. SIAM-ACM Symposium on Discrete Algorithms (SODA 10), SIAM Press, 2010, pp. 938-948.

[12] S. De Agostino, "LZW versus Sliding Window Compression on a Distributed System: Robustness and Communication," Proc. INFOCOMP, IARIA, 2011, pp. 125-130.

[13] S. De Agostino, "Low-Complexity Lossless Compression on High Speed Networks," Proc. ICSNC, IARIA, 2012, pp. 130-135.

[14] J. A. Storer and T. G. Szimansky, "Data Compression via Textual Substitution," Journal of ACM, vol. 24, 1982, pp. 928-951.

[15] T. A. Welch, "A Technique for High-Performance Data Compression," IEEE Computer, vol. 17, 1984, pp. 8-19.

[16] J. A. Storer, Data Compression: Methods and Theory, Computer Science Press, 1988.

[17] S. De Agostino, "Almost Work-Optimal PRAM EREW Decoders of LZ-Compressed Text," Parallel Processing Letters, vol. 14, 2004, pp. 351-359.

[18] S. De Agostino, "LZW Data Compression on Large Scale and Extreme Distributed Systems," Proceedings Prague Stringology Conference, 2012, pp. 18-27.

[19] S. De Agostino, "Bounded Memory LZW Compression and Distributed Computing: A Worst Case Analysis," Proceedings Festschrift for Borivoj Melichar, 2012, pp. 1-9.

 ISBN: 978-1-61208-280-6