The Uncompress Application on Distributed Communications Systems

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

***Abstract*—The most popular compressors are based on Lempel-Ziv coding methods. Zip compressors and Unzip decompressors apply the sliding window method, while other applications as Compress and Uncompress under Unix and Linux platforms use the so-called LZW compressor and decompressor. LZW compression is less effective but faster than the zipping applications. We face the problem of how to implement Lempel-Ziv data compression on today's large scale distributed communications systems. Zipping and unzipping files is parallelizable in theory. However, the number of global computation steps is not bounded by a constant and a local computation approach is more advantageous on a distributed system. Such approach might cause a lack of robustness when scalability properties are required. Differently from the Zip compressors, the LZW encoder/decoder presents an asymmetry with respect to global parallel computation since the encoder is not parallelizable while the decoder has a very efficient parallelization. We show that, in practice, the number of iterations of the LZW parallel decoder (Uncompress) is much less than ten units. Since scalability and robustness are generally guaranteed if bounding the number of global computation steps is possible, LZW is more attractive than Zip for distributed communications systems in those cases (which are the most common in practice) where compression is performed only once or very rarely, while the frequent reading of raw data needs fast decompression.**

Keywords-distributed application; communication; scalability; robustness; data compression

I. INTRODUCTION

The most popular compressors are based on Lempel-Ziv coding methods (LZ compression). Zip compressors and Unzip decompressors apply the sliding window method, while other applications as Compress and Uncompress under Unix and Linux platforms use the so-called LZW compressor and decompressor. LZW compression is less effective but faster than the zipping applications. LZ compression [1][2] is based on string factorization. Two different factorization processes exist with no memory constraints. With the first one (LZ1) [1], 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) [2], 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 theoretical parallel algorithms [3]-[6], LZ2 compression is hard to parallelize [7]. This difference is mantained when bounded memory versions of LZ compression are considered [5][6][8]. On the other hand, parallel decompression is possible for both approaches [4][9]. The Zip compressors implement a bounded memory version (sliding window) of LZ1, while LZW compression is a bounded memory variant of LZ2.

We face the problem of how to implement LZ data compression on today's large scale distributed communications systems. The computing techniques involved in the design of parallel and distributed algorithms strictly relate to the computational model on which the distributed communications system is based. The efficiency of a technique designed for a specific model can consistently deteriorates when applied to a different system. This is particularly evident when a technique designed for a shared memory parallel random access machine (PRAM) is implemented on a distributed memory system. Indeed, when the system is scaled up the communication cost is a bottleneck to linear speed-up. So, we need to limit the interprocessor communication either by involving more local computation or by bounding the number of global computation steps in order to obtain a practical algorithm. Local computation might cause a lack of robustness when scalability properties are required. On the other hand, scalability and robustness are generally guaranteed if bounding the number of global computation steps is possible for a specific problem.

As mentioned above, zipping and unzipping files are parallelizable in theory. However, the number of global computation steps is not bounded by a constant and the local computation approach is more advantageous on a distributed memory system. A distributed algorithm, approximating in practice the compression effectiveness of the Zip application, has been realized in [10] on an array of processor with no interprocessor communication. An approach using a tree architecture slightly improves compression effectiveness [11]. Yet, distributed algorithms approximating in practice the compression effectiveness of the Compress application have been realized in [12] with very low communication cost. However, scalability and robustness for each of these LZ compression distrbuted algorithms are guaranteed only on very large size files.

In this paper, we evidentiate that, differently from the Zip compressors, the LZW encoder/decoder presents an

 ISBN: 978-1-61208-404-6

asymmetry with respect to global parallel computation since the encoder is not parallelizable while the decoder has a very efficient parallelization. We show that, in practice, the number of iterations of the LZW parallel decoder is much less than ten units. This makes LZW more attractive than Zip for distributed communications systems in those cases (which are the most common in practice) where compression is performed very rarely while the frequent reading of raw data needs fast decompression.

In Section II, we describe the LZ data compression techniques and their bounded memory versions. In Section III, we discuss the parallel complexity of LZ data compression and decompression. Section IV shows how the implementation of the parallel LZW decompressor (Uncompress) is suitable for distributed communications systems. Conclusions and future work are given in Section V.

II. LZ Data Compression

LZ data compression is a dictionary-based technique. Indeed, 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.

A. 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 $f_1 f_2 \cdots f_{i-1}$ [13]. 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.

B. 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 all 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 [14]. f_i is encoded by a pointer q_i to such concatenation (unbounded LZW compression). Differently from LZ1 and LZSS, the dictionary is only prefix.

C. Bounded Size Dictionary Compression

The factorization processes described in the previous subsections 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 a fixed length code is used for the pointers. With sliding window compression, this can be simply obtained by using a fixed length window (therefore, the left end of the window slides as well) and by bounding the match length. Simple real time implementations are realized by means of hashing techniques providing a specific position in the window where a good appriximation of the longest match is found on realistic data. The window length is usually several kilobytes. The compression tools of the Zip family, as the Unix command gzip for example, use a window size of at least 32 Kb.

With LZW compression the dictionary elements are removed by using a deletion heuristic [15]. 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 either the longest match with the concatenation of a previous factor f_j, with $j \leq d$, and the next character or the current alphabet character if there is no match. 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 (*standard LZW encoding*). 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 (or the current alphabet character if there is no match since the restart operation removes all the elements from the dictionary but the alphabet characters). This heuristic is the one used by standard applications (as Compress under Unix and Linux platforms) 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 least recently used (LRU) 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 [8], a relaxed version (RLRUp) was introduced. RLRUp 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. RLRUp turns out to be as good as LRU even when p is equal to 2 [16]. Since RLRU2 removes an arbitrary element from the equivalence class with the "older" elements, the two classes can be implemented with a couple of stacks, which makes RLRU2 slightly easier to implement than LRU in addition to be more space efficient. SWAP is the best heuristic among the "discrete" ones. Each of the bounded versions of LZW compression, described in this subsection, can be implemented in real time by storing the dictionary in a trie data structure.

III. Parallel Complexity

The model of computation we consider in this section is the CREW (cuncurrent read, exclusive write) PRAM, that is, a parallel machine where processors access a shared memory without writing conflicts. As mentioned in the introduction, speed is more relevant with decompression than with compression since in most cases compression is performed very rarely while the frequent reading of raw data needs a fast decoder. Therefore, we briefly discuss the parallel complexity issues concerning compression and, then, describe the parallel decoders for sliding window compression and LZW compression.

A. Parallel Complexity of LZ Compression

LZSS (or LZ1) compression (that is, the zipping application) can be efficiently parallelized from a theoretical point of view [3]-[6]. On the other hand, LZW (or LZ2) compression is P-complete [7] and, therefore, hard to parallelize. Decompression, instead, is parallelizable for both methods [4][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 (that is, Compress), while the SWAP heuristic does not seem to have a parallel decoder [8]. Moreover, the SC^k-hardness 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 [8]. In [8], the RLRUp relaxed version was introduced in order to obtain the first (and only so far) natural SC^k-complete problem by partitioning the dictionary in p equivalence classes and by considering all the elements in each class to have the same "age" for the LRU strategy. As mentioned in the previous section, RLRU2 turns out to be as good as LRU and it is slightly easier to implement in addition to be more space efficient. In conclusion, the only practical LZW compression algorithm for a shared memory parallel system is the one using the FREEZE deletion heuristic [6].

B. Parallel Unzip

The design of a parallel decoder for sliding window compression (that is, the unzipping application) is based on a reduction to the problem of finding the trees of a forest in O(k) time with O(n/k) processors on a CREW PRAM, if k is $\Omega(\log n)$ and n is the number of nodes. Given the sequence of pointers $q_i = (d_i, \ell_i)$, for $1 \leq i \leq m$, produced by the application zipping an input file, let $s_1, ..., s_m$ be the partial sums of $l_1, ..., l_m$. Then, 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, O(n/k) processors store the parent pointers in an array of size n for blocks of k positions and apply the pointer jumping technique to find the trees.

C. Parallel Uncompress

We already pointed out that the decoding problem is interesting independently from the computational efficiency of the encoder. This is particularly evident in the case of compressed files stored in a ROM since 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 (Uncompress application). The positions of the special mark can be detected by parallel prefix.

Let $Q_1 \cdots Q_N$ be the standard LZW encoding of a string S, drawn over an alphabet A of cardinality α, with Q_h

 ISBN: 978-1-61208-404-6

sequence of pointers between two consecutive restart operations, for $1 \leq h \leq N$. Let D be the bounded size dictionary employed by the compression algorithm, with $d = |D|$. Each Q_h can be decoded independently. Let $q_1...q_m$ be the sequence of pointers Q_h encoding the substring S' of S. The decoding of Q_h can be parallelized on a CREW PRAM in $O((log(L))$ time with $O(|S'|)$ processors, where L is the maximum length of a pointer target. d is a theoretical upper bound to L, that is tight for unary strings. The algorithm works with an initially null m x d matrix M and applies to M the procedure of Figure 1 [9].

```
k = 1;
in parallel for 1 ≤ i ≤ m do begin
    M[1, i] := q_i;
    last[i] := 1;
    value[i] := M[1, i] – d;
    while value(i) > 0 do begin
        in parallel for 1 ≤ j ≤ k do
            if j ≤ last(value(i)) then M[last(i) + j, i] := M[j, value(i)];
        last(i) := last(i) + last(value(i));
        value(i) := value(value(i));
        k = 2k;
    end;
end
```

Figure 1. The CREW PRAM procedure.

At each step, $last[i]$ is the last nonnull component on the i^{th} column considered. The nonnull components of the $value(i)^{th}$ column are copied on the i^{th} column in the positions after $last[i]$. Note that $value(i)$ is strictly less than i. Then, $value(i)$ is updated by setting $value(i) := value(value(i))$. The iteration stops on the column i when $value(i)$ is less or equal to zero. This procedure takes $O(|S'|)$ processors and $O(log(L))$ time on a CREW PRAM, since the number of nonnull componenents on a column doubles at each step. 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 $q_i - \alpha + 1$, since the dictionary contains initially the alphabet characters. At the end of the procedure, $M[last[i], i]$ is the pointer representing the first character of the target of q_i and $last[i]$ is the target length. Then, we conclude that $M[last[M[j, i] - d + 1], M[j, i] - d + 1]$, for $1 \leq j \leq last[i] - 1$, is the pointer representing the $(last[i] - j + 1)^{th}$ character of the target of q_i. That is, we have to look at the pointer values written on column i and consider the last nonnull components of the columns in the positions given by such values decreased by $\alpha - 1$. Such components must be concatenated according to the bottom-up order of the respective values on column i. By mapping each component into the correspondent alphabet character, we obtain the suffix following the first character of the target of q_i and the pointers are, therefore, decoded.

IV. Distributed Communications Systems and The Uncompress Application

Shared memory machines, as the PRAM model, are ideal systems for distributed communications. Realistically, such systems are feasible with the current technology only when the scale is very limited (ten units is the order of magnitude). The scalability requirement implies that the memory of the system is distributed. However, the PRAM model might be useful for a first approach to the design of an algorithm for distributed communications systems. Before discussing such approach, we consider distributed memory systems with no or very low interprocessor communication cost during the computational phase and, then, we discuss the requirements that a model of computation must have in order to yield a practical algorithm for distributed communications systems. Finally, we discuss the implementation of Uncompress and compare it with Unzip.

A. Star and Extended Star Networks

Distributed memory 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, as mentioned in the introduction, we need to limit the interprocessor communication either by involving more local computation or by bounding the number of global computation steps in order to design a practical algorithm. If we consider the local computation approach. the simplest model is 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 hn 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 to scale up the system.

For every integer k greater than 1, we can apply in parallel sliding window compression to blocks of length kw with $O(n/kw)$ processors connected to a central node of a star network, where n and w are the lengths of the input string and the window respectively [10]. If the order of magnitude of the block length is greater than the one of the window length, the compression effectiveness of the distributed implementation is about the same as the sequential one on realistic data. Since the compression tools of the Zip family use a window size of at least 32 Kb, the

 ISBN: 978-1-61208-404-6

block length should be about 300 Kb and the file size should be about one third of the number of processors in megabytes. Therefore, the application is suitable only for a small scale system unless the file size is very large.

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. In order to speed up the static phase after the dictionary is filled up, an implementation is provided on an extended star topology in [12]. The Compress application employs a dictionary of size 2^{16} and works with the RESTART deletion heuristic (also, called LZC compression [17]). The block length needed to fill up a dictionary of this size is approximately 300 Kb and the scalability and robustness issues with Compress are the same as with Zip.

B. A Model of Computation

Distributed communications systems allow global computation and bounding the number of computational steps is a requirement for the design of a practical algorithm. In [18], necessary requirements for a practical model of distributed computation were proposed, by considering the MapReduce programming paradigm which is the most in fashion for the design of an application for distributed communications systems. The following complexity requirements are stated as necessary for a practical interest:

- the number of global computation steps is polylogarithmic in the input size n;
- the number of processors and the amount of memory for each processor are sublinear;
- at each step, each processor takes polynomial time.

In [18], it is also shown that a $t(n)$ time CREW PRAM algorithm using subquadratic work space and a subquadratic number of processors has an implementation satisfying the above requirements if $t(n)$ is polylogarithmic. Indeed, the number of global computation steps of the implementation is O($t(n)$) while the subquadratic work space is partitioned among a sublinear number of processors taking polynomial computational time. Such requirements are necessary but not sufficient to guarantee a speed-up of the computation. Obviously, the total running time cannot be higher than the sequential one and this is trivially implicit in what is stated in [18]. The non-trivial bottleneck is the communication cost of the computational phase. This needs to be checked experimentally since the number of global computation steps 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 with no interprocessor communication. Moreover, if we want the speed-up to be linear then the total running time of a processor must be O($t(n)/n^{1-\epsilon}$), where $t(n)$ is the sequential time and $n^{1-\epsilon}$ is the number of processors. These stronger requirements are satisfied by the distributed implementation of Zip and Unzip presented in the first subsection. The LZW encoder/decoder implemented on the extended star network has a low communication cost, which is still affordable. Based on a worst case analysis, a more robust approach with no interprocessor communication is presented in [19] for compressing very large size files, which uses the RLRU2 deletion heuristic.

Generally speaking, an application on distributed communications systems has a practical interest if the number of global computation steps is about ten units or less. This is obtained from the simulation of the CREW PRAM implementation of the Uncompress application, together with the other requirements mentioned above.

C. Uncompress versus Unzip with Pointer Jumping

Computing the trees of a forest for Unzip and, implicitly, computing for each tree of a forest the paths from the nodes to the root for Uncompress are the problems we faced with parallel decompression. If we use a parent array as data structure to represent a forest, these problems can easily be solved on a CREW PRAM by running in parallel the pointer jumping operation *parent*[i] = *parent*[*parent*[i]]. The procedure takes a number of processors linear in the number of nodes and a time logarithmic in the maximum height of a tree. However, while with the unzipping application the maximum height of a tree can reach the order of magnitude of the string length, Uncompress deals with very shallow trees. This means that we can parallelize the LZW decoder with many fewer iterations. Moreover, the number of children for each node is very limited in practice and cuncurrent reading can be easily managed by standard bdroadcasting techniques on today's available clusters.

Now, we are ready to discuss the complexity issues with respect to distributed communications systems of the parallelization of the Uncompress application presented in [9]. Standard LZW encoding applies the "restart" operation to a dictionary of size 2^{16} in the Unix and Linux Compress applications, as pointed out in the first subsection, and similar applications have been realized with Stuffit on Windows and Dos platforms. As previously mentioned, the theoretical upper bound to the factor length is the dictionary size, which is tight in the unary string case. However, on realistic data we can assume that the maximum factor length L is such that $10 < L < 20$. The motivation for this assumption is that, in practice, the maximum length of a factor is much smaller than the dictionary size. For example, when compressing english text with sixteen bits pointers, the average match length will only be about five units (for empirical results, see [15]). In some exceptional cases, the maximum factor length will reach one hundred units, that is, the number of iterations (global computation steps) will

 ISBN: 978-1-61208-404-6

be equal to seven units. If $Q^1 \cdots Q^N$ is the encoding of the input string S, with $Q^h = q_1^h ... q_{m^h}^h$ sequence of pointers between two consecutive restart operations for $1 \leq h \leq N$, let H =max$\{m^h : 1 \leq h \leq N\}$. It is easy to implement the parallel procedure on a cluster of H processors, where the input is a set $\cup_{h=1}^{N} V^h$ and each element in V^h is a pointer q_i^h, for $1 \leq i \leq m^h$ and $1 \leq h \leq N$. This set is distributed among the H processors so that the i-th processor receives pointer q_i^h (if it exists), for $1 \leq h \leq N$.

The sub-linearity requirements stated in [18] are satisfied, since N and m^h, for $1 \leq h \leq N$, are generally sub-linear in practice. Moreover, the running time multiplied by the number of processors is O(T), with T sequential time, since the number of global computation steps is about ten units (optimality requirement). This makes the implementation of practical interest.

V. Conclusion

In this paper, we presented an implementation of the Uncompress application on distributed communications systems. Although the Compress application is not parallelizable in practice, those characteristics implying hardness results with respect to the parallel complexity of the encoder are the same providing asymmetrically a practical parallel decoder. Since, in most cases, compression is performed only once or very rarely, while the frequent reading of raw data needs fast decompression, Compress can be considered attractive in a parallel fashion. Indeed, Unzip is much less practical to parallelize.

Parallel and distributed algorithms for LZ data compression and decompression is a field that 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. However, there is a lack of robustness of the practical compression distributed algorithms when the system is scaled up and the order of magnitude of the file size is smaller than one gigabyte. As long as the communication cost is a relevent bottleneck of current technology, considering Compress and focusing only on speeding up Uncompress has good motivations since there is still a need for compression for purposes both of storage and transmission when the file size is a hundred megabytes or less and a novel application speeding up decoding requires robustness in such cases. As future work, we would like to implement Uncompress on today's large scale commodity clusters.

References

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

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

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

[4] M. Farach and S. Muthikrishnan, *Optimal Parallel Dictionary Matching and Compression,* Proceedings SPAA, 1995, pp. 244-253.

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

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

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

[8] 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.

[9] S. De Agostino. *A Parallel Decoding Algorithm for LZ2 Data Compression,* Parallel Computing, vol. 21, 1995, pp. 1957-1961.

[10] 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.

[11] S. T. Klein and Y. Wiseman, *Parallel Lempel-Ziv Coding,* Discrete Applied Mathematics, vol. 146, 2005, pp. 180-191.

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

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

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

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

[16] 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.

[17] T. C. Bell, J. G. Cleary and I. H. Witten, *Text Compression,* Prentice Hall, 1990.

[18] 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.

[19] S. De Agostino, *A Robust Approach to Large Size Files Compression using the MapReduce Web Computing Framework,* International Journal on Advances in Internet Technology, vol. 7, 2014, pp. 29-38.

 ISBN: 978-1-61208-404-6