LZW versus Sliding Window Compression on a Distributed System: Robustness and Communication

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

***Abstract*—Scalability preserves the robustness of sliding window compression only on very large files when it is implemented on a distributed system with low communication cost. On the other hand, we show that Lempel-Ziv-Welch compression is scalable and robust on arbitrary files.**

***Keywords*-dictionary-based compression, string factorization, parallel complexity, distributed algorithm.**

I. Introduction

Lempel-Ziv 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 (sliding window compression). 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 (Lempel-Ziv-Welch or LZW compression). This computational difference implies that while sliding window compression has efficient parallel algorithms [4], [5] LZW compression is hard to parallelize [6]. This difference is mantained when bounded memory versions of Lempel-Ziv compression are considered [5], [7], [8]. On the other hand, parallel decompression is possible for both approaches [10]. This field has developed in the last twenty years from a theoretical approach concerning parallel time complexity with no memory constraints to the practical goal of designing distributed algorithms with bounded memory and low communication cost. While with shared memory machines scalability is always possible [11], this is not always garanteed with distributed memory. 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. An example of distibuted system with low communication cost is a tree architecture. Distributed algorithms for sliding window compression approximating in practice its compression effectiveness has been realized in [8] on an array of processor with no interprocessor communication. An approach using a tree architecture slightly improves compression effectiveness [9]. However, the scalability of a parallel implementation of sliding window compression on a distributed system with low communication cost garantees robustness only on very large files. On the other hand, we show in this paper that LZW compression is scalable and robust on arbitrary files if implemented on a tree architecture.

In Section 2, we describe the Lempel-Ziv compression techniques and in Section 3, we present the bounded memory versions. In Section 4, we present previous work ona parallel system with shared memory. Section 5 discuss how Lempel-Ziv data compression and decompression can be implemented on a distributes system and compare LZW compression with the sliding window technique. Conclusions and future work are given in Section 6.

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 (sliding window) compression is also called the sliding dictionary method while LZ2 (LZW) compression is more generally called the dynamic dictionary method.

A. Sliding Window 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 $f_1 f_2 \cdots f_{i-1}$ [12]. f_i is encoded by the pointer $q_i = (d_i, \ell_i)$, where d_i is the displacement back to the copy of the factor and ℓ_i is the length of the factor (LZSS compression). If $d_i = 0$, l_i is the alphabet character. In other words the dictionary is defined by a window sliding its right end over the input string, that is, it comprises all the substrings of the prefix read so far in the computation. It follows that the dictionary is both *prefix* and *suffix* since all the prefixes and suffixes of a dictionary element are dictionary elements. The position of the longest match in the prefix with the current position can be computed in real time by means of a suffix tree data structure [13], [14].

B. LZW Compression

The LZ2 factorization of a string S is $S = f_1 f_2 \cdots f_i \cdots f_k$ where f_i is the shortest substring, which is different from one of the previous factors. As for LZ1 the encoding of each factor leaves one character uncompressed. To avoid this a different factorization was introduced (LZW factorization) where each factor f_i is the longest match with the concatenation of a previous factor and the next character [15]. f_i is encoded by a pointer q_i to such concatenation (LZW compression). LZW compression can be implemented in real time by storing the dictionary with a trie data structure. Differently from sliding window compression, the dictionary is only prefix.

C. Greedy versus Optimal Factorization

The pointer encoding the factor f_i has a size increasing with the index i. This means that the lower is the number of factors for a string of a given length the better is the compression. The factorizations described in the previous subsections are produced by greedy algorithms. The question is whether the greedy approach is always optimal, that is, if we relax the assumption that each factor is the longest match can we do better than greedy? The answer is negative with suffix dictionaries as for sliding window compression. On the other hand, the greedy approach is not optimal for 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 [17].

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. 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 LZW compression dictionary elements are removed by using a deletion heuristic. The deletion heuristics we describe in this section are FREEZE, RESTART and LRU [18]. Then, we give more details on sliding window compression.

A. 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 “freezed”. Afterwards, the factorization continues in a “static” way using the factors of the freezed dictionary. In other words, the LZW factorization of a string S using the FREEZE deletion heuristic is $S = f_1 f_2 \cdots f_i \cdots f_k$ where f_i is the longest match with the concatenation of a previous factor f_j, with $j \leq d$, and the next character. The shortcoming of this heuristic is that after processing the string for a while the dictionary often becomes obsolete. A more sophisticated deletion heuristic is RESTART, which monitors the compression ratio achieved on the portion of the imput string read so far and, when it starst deteriorating, restarts the factorization process. Let $f_1 f_2 \cdots f_j \cdots f_i \cdots f_k$ be such factorization with j the highest index less than i where the restart operation happens. Then, f_j is an alphabet character and f_i is the longest match with the concatenation of a previous factor f_h, with $h \geq j$, and the next character (the restart operation removes all the elements from the dictionary but the alphabet characters). This heuristic is used by the Unix command Compress since it has a good compression effectiveness and it is easy to implement. However, the best deletion heuristic is LRU (last recently used strategy). The LRU deletion heuristic removes elements from the 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.

B. 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. A real time implementation of compression with finite window is possible using a suffix tree data structure [19]. Much simpler 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. In [20], the three current characters are hashed to yield a pointer into the already compressed text. In [21], hashing of strings of all lengths is used to find a match. In both methods, collisions are resolved by overwriting. In [22], the two current characters are hashed and collisions are chained via an offset array. Also the Unix gzip compressor chains collisions but hashes three characters [23].

C. Greedy versus Optimal Factorization

Greedy factorization is optimal for compression with finite windows since the dictionary is suffix. With LZW compression, after we fill up the dictionary using the FREEZE or RESTART heuristic, the greedy factorization we compute with such dictionary is not optimal since the dictionary is not suffix. However, there is an optimal semi-greedy factorization, which at each step computes a factor such that the longest match in the next position with a dictionary element ends to the rightest [24], [25]. Since the dictionary is prefix, the factorization is optimal. The algorithm can even be implemented in real time with a modified suffix tree data structure [24].

IV. Previous Work

Sliding window compression can be efficiently parallelized on a PRAM EREW [4], [5], [8], that is, a parallel machine where processors access a shared memory without reading and writing conflicts. On the other hand, LZW compression is P-complete [6] and, therefore, hard to parallelize. Decompression, instead, is parallelizable for both methods [10]. As far as bounded size dictionary compression is concerned, the "parallel computation thesis" claims that sequential work space and parallel running time have the same order of magnitude giving theoretical underpinning to the realization of parallel algorithms for LZW compression using a deletion heuristic. However, the thesis concerns unbounded parallelism and a practical requirement for the design of a parallel algorithm is a limited number of processors. A stronger statement is that sequential logarithmic work space corresponds to parallel logarithmic running time with a polynomial number of processors. Therefore, a fixed size dictionary implies a parallel algorithm for LZW compression satisfying these constraints. Realistically, the satisfaction of these requirements is a necessary but not a sufficient condition for a practical parallel algorithm since the number of processors should be linear, which does not seem possible for the RESTART deletion heuristic. Moreover, the SC^k-completeness of LZ2 compression using the LRU deletion heuristic and a dictionary of polylogarithmic size shows that it is unlikely to have a parallel complexity involving reasonable multiplicative constants [7]. In conclusion, the only practical LZW compression algorithm for a shared memory parallel system is the one using the FREEZE deletion heuristic. We will see these arguments more in details in the next subsections.

A. Sliding Window Compression on a Parallel System

We present compression algorithms for sliding dictionaries on an exclusive read, exclusive write shared memory machine requiring $\mathrm{O}(k)$ time with $\mathrm{O}(n/k)$ processors if k is $\Omega(\log n)$, with the practical and realistic assumption that the dictionary size and the match length are constant [8]. As previously mentioned, greedy factorization is optimal with sliding dictionaries. In order to compute a greedy factorization in parallel we find the greedy match in each position i of the input string and link i to $j+1$, where j is the last position of the match. If the greedy match ends the string i is linked to $n+1$, where n is the length of the string. It follows that we obtain a tree rooted in $n+1$ and the positions of the factors are given by the path from 1 to $n+1$. Such tree can be built in $\mathrm{O}(k)$ time with $\mathrm{O}(n/k)$ processors. In fact, on each block of k positions one processor has to compute a match having constant length and the reading conflicts with other processors are solved in logarithmic time by standard broadcasting techniques. Then, since for each node of the tree the number of children is bounded by the constant match length it is easy to add the links from a parent node to its children in $\mathrm{O}(k)$ time with $\mathrm{O}(n/k)$ processors and apply the well-known Euler tour technique to this doubly linked tree structure to compute the path from 1 to $n+1$.

B. The Completeness Results

NC is the class of problems solvable with a polynomial number of processors in polylogarithmic time on a parallel random access machine and it is comjectured to be a proper subset of P, the class of problems solvable in sequential polynomial time. LZ2 and LZW compression with an unbounded dictionary have been proved to be P-complete [6] and, therefore, hard to parallelize. SC is the class of problems solvable in polylogarithmic space and sequential polynomial time. The LZ2 algorithm with LRU deletion heuristic on a dictionary of size $\mathrm{O}(\log^k n)$ can be performed in polynomial time and $\mathrm{O}(\log^k n \log\log n)$ space, where n is the length of the input string. In fact, the trie requires $\mathrm{O}(\log^k n)$ space by using an array implementation since the number of children for each node is bounded by the alphabet cardinality. The $\log\log n$ factor is required to store the information needed for the LRU deletion heuristic since each node must have a different age, which is an integer value between 0 and the dictionary size. Obviously, this is true for the LZW algorithm, as well. If the size of the dictionary is $\mathrm{O}(\log^k n)$, the LRU strategy is log-space hard for SC^k, the class of problems solvable simultaneously in polynomial time and $\mathrm{O}(\log^k n)$ space [7]. The problem belongs to SC^{k+1}. This hardness result is not so relevant for the space complexity analysis since $\Omega(\log^k n)$ is an obvious lower bound to the work space needed for the computation. Much more interesting is what can be said about the parallel complexity analysis. In [7], it was shown that LZ2 (or LZW) compression using the LRU deletion heuristic with a dictionary of size c can be performed in parallel either in $\mathrm{O}(\log n)$ time with $2^{\mathrm{O}(c\log c)}n$ processors or in $2^{\mathrm{O}(c\log c)}\log n$ time with $\mathrm{O}(n)$ processors. This means that if the dictionary size is constant, the compression problem belongs to NC. NC and SC are classes that can be viewed in some sense symmetric and are believed to be incomparable. Since log-space reductions are in NC, the compression problem cannot belong to NC when the dictionary size is polylogarithmic if NC and SC are incomparable. We want to point out that the dictionary size c figures as an exponent in the parallel complexity of the problem. This is not by accident. If we believe that SC is not included in NC, then the SC^k-

hardness of the problem when c is $\mathrm{O}(\log^k n)$ implies the exponentiation of some increasing and diverging function of c. In fact, without such exponentiation either in the number of processors or in the parallel running time, the problem would be SC^k-hard and in NC when c is $\mathrm{O}(\log^k n)$. Observe that the P-completeness of the problem, which requires a superpolylogarithmic value for c, does not suffice to infer this exponentiation since c can figure as a multiplicative factor of the time function. Moreover, this is a unique case so far where somehow we use hardness results to argue that practical algorithms of a certain kind (NC in this case) do not exist because of huge multiplicative constant factors occurring in their analysis. In [7], a relaxed version (RLRU) was introduced, which turned out to be the first (and only so far) natural SC^k-complete problem.

C. *LZW Compression on a Parallel System*

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

D. *Parallel Deompression*

The design of parallel decoders is based on the fact that the Euler tour technique can also be used to find the trees of a forest in $\mathrm{O}(k)$ time with $\mathrm{O}(n/k)$ processors on a shared memory parallel machine without writing and reading conflicts, if k is $\Omega(\log n)$ and n is the number of nodes. We present decoders paired with the practical coding implementations using bounded size dictionaries. First, we see how to decode the sequence of pointers $q_i = (d_i, \ell_i)$ produced by the sliding window method with $1 \leq i \leq m$ [10]. If $s_1, ..., s_m$ are the partial sums of $l_1, ..., l_m$, the target of q_i encodes the substring over the positions $s_{i-1}+1 \cdots s_i$ of the output string. Link the positions $s_{i-1}+1 \cdots s_i$ to the positions $s_{i-1}+1-d_i \cdots s_{i-1}+1-d_i+l_i-1$, respectively. If $d_i = 0$, the target of q_i is an alphabet character and the corresponding position in the output string is not linked to anything. Therefore, we obtain a forest where all the nodes in a tree correspond to positions of the decoded string where the character is represented by the root. The reduction from the decoding problem to the problem of finding the trees in a forest can be computed in $\mathrm{O}(k)$ time with $\mathrm{O}(n/k)$ processors where n is the length of the output string, because this is the complexity of computing the partial sums since $m \leq n$. Afterwards, one processor stores the parent pointers in an array of size n for a block of k positions. We can make the forest a doubly linked structure since the window size is constant and apply the Euler tour technique to find the trees. With LZW compression using the FREEZE deletion heuristic the parallel decoder is trivial. We wish to point out that the decoding problem is interesting independently from the computational efficiency of the encoder. In fact, in the case of compressed files stored in a ROM only the computational efficiency of decompression is relevant. With the RESTART deletion heuristic, a special mark occurs in the sequence of pointers each time the dictionary is cleared out so that the decoder does not have to monitor the compression ratio. The positions of the special mark are detected by parallel prefix. Each subsequence $q_1 \cdots q_m$ of pointers between two consecutive marks can be decoded in parallel but the pointers do not contain the information on the length of their targets and it has to be computed. The target of the pointer q_i in the subsequence is the concatenation of the target of the pointer in position $q_i - \alpha$ with the first character of the target of the pointer in position $q_i - \alpha + 1$, where α is the alphabet cardinality. Then, in parallel for each i, link pointer q_i to the pointer in position $q_i - \alpha$, if $q_i > \alpha$. Again, we obtain a forest where each tree is rooted in a pointer representing an alphabet character and the length l_i of the target of a pointer q_i is equal to the level of the pointer in the tree plus 1. It is known from [1] that the largest number of distinct factors whose concatenation forms a given string of length ℓ is $\mathrm{O}(\ell/\log \ell)$. Since a factor of the LZW factorization of a string appears a number of times, which is at most equal to the alphabet cardinality, it follows that m is $\mathrm{O}(\ell/\log \ell)$ if ℓ is the length of the substring encoded by the subsequence $q_1 \cdots q_m$. Then, building such a forest takes $\mathrm{O}(k)$ time with $\mathrm{O}(n/k)$ processors on a shared memory parallel machine without writing and reading conflicts if k is $\Omega(\log n)$. By means of the Euler tour technique, we can compute the trees of such forest and the level of each node in its own tree in $\mathrm{O}(k)$ time with $\mathrm{O}(n/k)$. Therefore, we can compute the lengths $l_1, ..., l_m$ of the targets. If $s_1, ..., s_m$ are the partial sums, the target of q_i is the substring over the positions $s_{i-1}+1 \cdots s_i$ of the output string. For each q_i, which does not correspond to an alphabet character, define $first(i) = s_{q_i-\alpha-1} + 1$ and $last(i) = s_{q_i-\alpha} + 1$. Since the target of the pointer q_i is the concatenation of the target of the pointer in position $q_i - \alpha$ with the first character of the target of the pointer in position $q_i - \alpha + 1$, link the positions $s_{i-1} + 1 \cdots s_i$ to the positions $s_{first(i)} \cdots s_{last(i)}$, respectively. As in the sliding dictionary case, if the target of q_i is an alphabet character the corresponding position in the output string is the root of a tree in a forest and all the nodes in a tree correspond to positions of the decoded string where the character is the root. Since the number of children for each node is at most α, in O(k) time and O(n/k) processors we can store the forest in a doubly linked structure and decode by means of the Euler tour technique [10].

V. LZW versus Sliding Window Compression on a Distributed System

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

A. Sliding Window Compression on a Distributed System

We simply apply in parallel sliding window compression to blocks of length kw. It follows that the algorithm requires O(kw) time with n/kw processors and the multiplicative approximation factor is $(k+m-1))/k$ with respect to any parsing. In fact, the number of factors of an optimal (greedy) factorization on a block is at least kw/m while the number of factors of the factorization produced by the scheme is at most $(k-1)w/m+w$. The boundary might cut a factor and the length w of the initial full size window of the block 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, 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 and the multiplicative approximation factor follows. We obtain an approximation scheme, which is suitable for a small scale system but due to its adaptiveness it works on a large scale parallel system when the file size is large. From a practical point of view, we can apply something like the gzip procedure to a small number of input data blocks achieving a satisfying degree of compression effectiveness and obtaining the expected speed-up on a real parallel machine. Making the order of magnitude of the block length greater than the one of the window length largely beats the worst case bound on realistic data and garantees robustness. 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. 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.

B. LZW Compression on a Distributed System

As mentioned at the beginning of this section, if we use a RESTART deletion heuristic clearing out the dictionary every ℓ characters of the input string we can trivially parallelize the factorization process with an O(ℓ) time, O(n/ℓ) processors distributed algorithm. LZW compression with the RESTART deletion heuristic was initially presented in [15] with a dictionary of size 2^{12} and is employed by the Unix command "compress" with a dictionary of size 2^{16}. Therefore, in order to have a satisfying compression effectiveness the distributed algorithm might work with blocks of length ℓ even greater than 300K on realistic data. After a dictionary is filled up for each block though, the factorization of the remaining suffix of the block can be approximated within the multiplicative factor $(k+1)/k$ in time O(km) with O(n/km) processors on a tree architecture. Every leaf processor stores a sub-block of length $m(k+2)$ and a copy of the dictionary, which are broadcasted from some level of the tree where the first phase of the computation has been executed. Adjacent sub-blocks overlap on $2m$ characters. We call a *boundary match* a factor covering positions of two adjacent sub-blocks. We execute the following algorithm:

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

Stopping the factorization of each sub-block at the beginning of the right boundary match might cause the making of a surplus factor, which determines the multiplicative approximation factor $(k+1)/k$ with respect to any factorization. In fact, the factor in front of the right boundary match might be extended to be a boundary match itself and to cover the first position of the factor after the boundary. In [26], it is shown experimentally that for $k = 10$ the compression ratio achieved by such factorizarion is about the same as the sequential one. Then, compression is effective and robust on a large scale system even if the size of the file is not large.

C. Decompression on a Distributed System

To decode the compressed files on a distibuted system, it is enough to use a special mark occuring 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 on a large scale tree architecture, a second special mark indicates for each block the end of the coding of a sub-block and the coding of each block is stored at the same level of the tree. The first sub-block for each block is decoded by one processor to learn the corresponding dictionary. Then, the subsequences of pointers coding the sub-blocks are broadcasted to the leaf processors with the corresponding dictionary.

VI. Conclusion

In this paper, we showed that with the low communication cost of a tree architecture we can scale up the implementation of LZW compression on a distributed system preserving its robustness. This does not seem to be possible with sliding window compression. As future work, we would like to implement Lempel-Ziv compression on available distributed systems as array and tree architectures.

References

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

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

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

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

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

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

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

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

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

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

[11] R. P. Brent, *The Parallel Evaluation of General Arithmetic Expressions,* Journal of the ACM 21, 201-206, 1974.

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

[13] M. Rodeh, V. R. Pratt, and S. Even, *Linear Algorithms for Compression via String Matching,* Journal of ACM 28, 16-24, 1980.

[14] E. M. Mc Creight, *A Space-Economical Suffix Tree Construction Algorithm,* Journal of ACM 23, 262-272, 1976.

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

[16] S. De Agostino and J. A. Storer, *On-Line versus Off-line Computation for Dynamic Text Compression,* Information Processing Letters 59, 169-174, 1996.

[17] S. De Agostino and R. Silvestri, *A Worst Case Analisys of the LZ2 Compression Algorithm,* Information and Computation 139, 258-268, 1997.

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

[19] E. R. Fiala and D. H. Green, *Data Compression with Finite Windows,* Communications of ACM 32, 490-505, 1988.

[20] J. R. Waterworth, *Data Compression System,* US Patent 4 701 745, 1987.

[21] R. P. Brent, *A Linear Algorithm for Data Compression,* Australian Computer Journal 19, 64-68, 1987.

[22] D. A. Whiting, G. A. George, and G. E. Ivey, *Data Compression Apparatus and Method,* US Patent 5016009, 1991.

[23] J. Gailly and M. Adler, http://www.gzip.org, 1991.

[24] A. Hartman and M. Rodeh, *Optimal Parsing of Strings.* In: Apostolico, A., Galil, Z. (eds.) Combinatorial Algorithms on Words, 155-167, Springer, 1985.

[25] M. Crochemore and W. Rytter, *Jewels of Stringology,* World Scientific, 2003.

[26] D. Belinskaya, S. De Agostino, and J. A. Storer, *Near Optimal Compression with respect to a Static Dictionary on a Practical Massively Parallel Architecture,* Proceedings IEEE Data Compression Conference, 172-181, 1995.