A Robust Approach to Large Size Files Compression using the MapReduce Web Computing Framework

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 on the basis of a theoretical worst case analysis, which evidentiates its robustness.**

Keywords-web computing; mapreduce framework; lossless compression; string factorization; worst case analysis

I. Introduction

Lempel-Ziv (LZ) techniques are the most widely used for lossless file compression and preliminary results on the distributed implementation, shown in this paper, were presented in [1], [2], [3]. LZ compression [4], [5], [6] is based on string factorization. Two different factorization processes exist with no memory constraints. With the first one (LZ1) [5], 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) [6], 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 [7], [8], [9], [10], LZ2 compression is hard to parallelize [11] 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 [9], [12]. 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 [12], 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) [12]. RLRU was shown to be as effective as LRU in [13] 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 [14], 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 [9], [15], [16], 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 II and III, we describe the Lempel-Ziv compression techniques and their bounded memory versions respectively. Section IV 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 V, 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 VI makes a comparative analysis of different implementations of LZ compression in this framework. A worst case analysis of standard LZ2 compression is given in Section VII and an alternative to the standard versions is proposed as the most efficient one for large size files compression. Conclusions and future work are given in Section VIII.

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.

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 $f_1 f_2 \cdots f_{i-1}$ [17]. 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 [18], [19].

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 every previous factor. 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 [20]. 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.

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 LZ1 or LZSS compression. On the other hand, the greedy approach is not optimal for LZ2 or LZW compression. However, the optimal approach is NP-complete [21] and the greedy algorithm approximates with an $O(n^{\frac{1}{4}})$ multiplicative factor the optimal solution [22].

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 LZSS (or LZ1) compression this can be simply obtained by sliding a fixed length window and by bounding the match length, for LZW (or LZ2) compression dictionary elements are removed by using a deletion heuristic. The deletion heuristics we describe in this section are FREEZE, RESTART, SWAP, LRU [23] and RLRU [12]. 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 "frozen". Afterwards, the factorization continues in a "static" way using the factors of the frozen dictionary. In other words, the LZW factorization of a string S using the FREEZE deletion heuristic is $S = f_1 f_2 \cdots f_i \cdots f_k$ where f_i is the longest match with the concatenation of a previous factor f_j, with $j \leq d$, and the next character. The shortcoming of this heuristic is that after processing the string for a while the dictionary often becomes obsolete. A more sophisticated deletion heuristic is RESTART, which monitors the compression ratio achieved on the portion of the 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. 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 LRU (last recently used strategy). The LRU deletion heuristic removes elements from the dictionary in a continuous way by deleting at each step of the factorization the least recently used factor that is not a proper prefix of another one. In [12], 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 [13]. 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.

B. Compression with Finite Windows

As mentioned at the beginning of this section, LZSS (or LZ1) bounded size dictionary compression is obtained by sliding a fixed length window and by bounding the match length. A real time implementation of compression with finite window is possible using a suffix tree data structure [24]. 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 [25], the three current characters are hashed to yield a pointer into the already compressed text. In [26], hashing of strings of all lengths is used to find a match. In both methods, collisions are resolved by overwriting. In [27], 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 [28].

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 is computed by the procedure of Figure 1 [29], [30]. At each step, we select a factor such that the longest match in the next position with a dictionary element ends to the rightest. Since the dictionary is prefix, the factorization is optimal. The algorithm can even be implemented in real time with an augmented trie data structure [29].

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

Figure 1. The semi-greedy factorization procedure.

IV. LZ Compression on a Parallel System

LZSS (or LZ1) compression can be efficiently parallelized on a PRAM EREW [7], [8], that is, a parallel machine where processors access a shared memory without reading and writing conflicts. On the other hand, LZW (or LZ2) compression is P-complete [11] and, therefore, hard to parallelize. Decompression, instead, is parallelizable for both methods [31]. 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 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 [12]. 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 [14] 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 $\mu_0\mu_1\rho_1 \cdots \mu_R\rho_R\mu_{R+1}\rho_{R+1}$, where (λ, x) is the unique $(key, value)$ pair input to μ_0 with λ the default initial (and final) key and x the input data. μ_0 distributes such data generating the multiset U_0 (μ_1 is the identity function or can be seen as a second step of the input phase). 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) with y output data.

The following complexity requirements are stated as necessary for a practical interest in [14]:

- 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 for each node is O($n^{1-\epsilon}$);
- mappers and reducers take polynomial time 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 [14]. 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 in O(ℓ) time with O(n/ℓ) processors, using any of the bounded memory compression techniques. Such distributed algorithms are suitable for a small scale system but due to their adaptiveness, they work on a large scale parallel system only when the file size is large.

A. Sliding Window Compression in MapReduce

With the sliding window method, ℓ is equal to kw where k is a positive integer and w is the window length [9], [15], [16]. 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 bytes. 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. It follows that the block length in our parallel implementation should be about 300 kB 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 $\mu_0\mu_1\rho_1\mu_2\rho_2$ where μ_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 $\ell \geq 300$ kB for $1 \leq i \leq m$. The complexity requirements of the MapReduce model are satisfied by the fact that ℓ is allowed to be strictly greater than 300 kB. The input to μ_0 is the pair $(0, X)$ mapping this element to the set U_0 of pairs $(1, X_1) \cdots (m, X_m)$. 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.

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 among the processors the subsequences of pointers coding each block.

B. LZW Compression Distributed Algorithms

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 [20]. The Unix command "compress" employs a dictionary of size 2^{16} and works with the RESTART deletion heuristic. The block length needed to fill up a dictionary of this size is approximately 300 kB. As previously mentioned, the SWAP heuristic is the best deletion heuristic among the discrete ones. After a dictionary is filled up on a block of 300 kB, the SWAP heuristic shows that we can use it efficiently on a successive block of about the same dimension, where a second dictionary is learned. A distributed compression algorithm employing the SWAP heuristic learns a different dictionary on every block of 300 kB of a partitioned string (the first block is compressed while the dictionary is learned). For the other blocks, block i is compressed statically in a second phase using the dictionary learned during the first phase on block $i-1$. But, unfortunately, the decoder is not parallelizable since the dictionary to decompress block i is not available until the previous blocks have been decompressed. On the other hand, with RESTART we can work on a block of 600 kB 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 300 kB 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 an extended star network (a tree of height 2) in time $O(km)$ with $O(n/km)$ processors, where k is a positive integer and m is the maximum factor length [2], [15].

In [15], during the input phase the central node of the extended star (that is, the root of the tree) broadcasts a block of length 600 kB to each adjacent processor. Then, for each block the corresponding processor broadcasts to the adjacent leaves a sub-block of length $m(k+2)$ in the suffix of length 300 kB, except for the first one and the last one which are $m(k+1)$ long. Each sub-block overlaps on m characters with the adjacent sub-block to the left and to the right, respectively (obviously, the first one overlaps only to the right and the last one only to the left). Every processor stores a dictionary initially set to comprise only the alphabet characters. The first phase of the computation is executed by processors adjacent to the central node. The prefix of length 300 kB of each block is compressed while learning the dictionary. At each step of the LZW factorization process, each of these processors sends the current factor to the adjacent leaves. They all adds such factor to their own dictionary. After compressing the prefix of length 300 kB of each block, all the leaves have a dictionary stored which has been learned by their parents during such compression phase.

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

- for each block, every corresponding leaf processor but the one associated with the last sub-block computes the boundary match between its sub-block and the next one ending furthest to the right, if any;
- each leaf processor computes the optimal factorization from the beginning of its sub-block to the beginning of the boundary match on the right boundary of its sub-block (or the end of its sub-block if there is no boundary match).

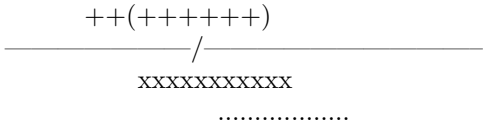

Figure 2. The making of a surplus factor.

Stopping the factorization of each sub-block at the beginning of the right boundary match might cause the making of a surplus factor, which determines the approximation factor $(k+1)/k$ with respect to any factorization. Indeed, as it is shown in Figure 2, the factor in front of the right boundary match (sequence of x's) might be extended to be a boundary match itself (sequence of plus signs) and to cover the first position of the factor after the boundary (dotted line).

In [32], it is shown experimentally that for $k = 10$ the compression ratio achieved by such factorizarion is about the same as the sequential one. Results were presented for static prefix dictionary compression but they are valid for dynamic compression using the LZW technique with the RESTART deletion heuristic. Indeed, experiments were realised compressing similar files in a collection using a dictionary learned from one of them. This is true even if the second step is greedy, since greedy is very close to optimal in practice. Moreover, with the greedy approach it is enough to use a simple trie data structure for the dictionary rather than the augmented suffix trie data structure of [29] needed to implement the semi-greedy factorization in real time. Therefore, in [2] after computing the boundary matches the second part of the parallel approximation scheme was substituted by the following procedure:

- each leaf processor computes the static greedy factorization from the end of the boundary match on the

left boundary of its sub-block to the beginning of the boundary match on the right boundary.

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

With standard large scale systems the sub-block length is several kilobytes with just a few megabytes to compress and the approach using boundary matches is too conservative for the static phase. In fact, a partition of the second half of the block does not effect on the compression effectiveness unless the sub-blocks are very small since the process is static. In conclusion, we proposed in [2] a further simplification of the algorithm for standard small, medium and large scale distributed systems.

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

$B_1 \cdots B_m$ is the sequence of blocks of length 600 kB partitioning the input file. Denote with B_i^1 and B_i^2 the two halves of B_i for $1 \leq i \leq m$. Divide B_i^2 into t sub-blocks of equal length. The input phase of this simpler algorithm distributes for each block the first half and the sub-blocks of the second half in the following way:

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

Then, the computational phase is:

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

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

As for the sliding window method, decoding the compressed file on a distributed system requires the presence of a special mark occurring in the sequence of pointers each time the coding of a block ends. The input phase distributes the subsequences of pointers coding each block among the processors. If the file is encoded by an LZW compressor implemented with one of the two procedures described in this subsection, a second special mark indicates for each block the end of the coding of a sub-block. The coding of the first half of each block is stored in one of the neighbors of the central node while the coding of the sub-blocks are stored into the corresponding leaves. The first half of each block is decoded by one processor to learn the corresponding dictionary. Each decoded factor is sent to the corresponding leaves during the process, so that the leaves can rebuild the dictionary themselves. Then, the dictionary is used by the leaves to decode the sub-blocks of the second half.

C. LZW Compression in MapReduce

In the MapReduce framework, the program sequence could be $\mu_0\mu_1\rho_1\mu_2\rho_2\mu_3\rho_3$ where $\mu_0\mu_1$ 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 $\ell \geq 300$ kB 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 $\ell' \geq 1000$ 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 300 kB and ℓ' strictly greater than 1000 bytes. Keys are pairs of positive integers. The input to μ_0 is the pair $((0,0), X)$, which is mapped to the set U_0 of pairs $((0,1), X_1Y_1)$, $\cdots$, $((0,m), X_mY_m))$, as input to μ_1. Then, μ_1 maps U_0 to the set U_0' of pairs $((0,1), X_1)$, $((1,1), B_{1,1})$, $\cdots$, $((1,r), B_{1,r})$, $\cdots$, $((0,m), X_m))$, $((m,1), B_{m,1})$, $\cdots$, $((m,r), B_{m,r})$. 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 program described does not compute boundary matches since we assumed the length of the sub-blocks to be at least 1000 bytes. When the length is between a hundred and a thousand bytes, the mapper μ_1 distributes overlapping sublocks and the reducer ρ_2 computes the boundary matches before completing the factorization process.

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 300 kB 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. This is also true for decompression, since the decoder is symmetrical as explained in the previous subsection.

D. A Comparative Analysis

We have described four different implementations of Lempel-Ziv data compression with the MapReduce framework. One implementation uses the sliding window technique while the other three are variants of the LZW compressor. The distributed implementations have irrelevant communication cost during the computational phase and keep the same characteristics of the sequential one on a single block of the distributed data. Therefore, LZW compression is less effective but faster than sliding window compression. In order to improve the effectiveness of LZW compression, the length of a single block of the distributed data is twice the one of the sliding window implementation. This can be done since the higher speed of the LZW compressor is kept in virtue of the fact that the compression of the second half of the block is not adaptive. Therefore, the distributed system can be arbitrarily scaled up when the second half is processed and there is no relevant slow-down. The first of the three distributed implementations proposed for the LZW compressor has a preprocessing phase and a nearly-optimal approach to the compression of the second half of the block. However, we observe with the second implementation that we can relax on the quasi-optimality of the approach since a left to right greedy algorithm performs well in practice. Finally, we notice that the preprocessing phase is needed only if the size of the distributed system is beyond standard large scale and a third implementation for standard large scale systems is presented, which is almost as simple as the one for the sliding window technique.

VII. LZW Compression and Worst Case Analysis

The approaches to LZW compression described above are not robust when the data are highly disseminated [3]. However, when compressing large size files even on a large scale system the size of the blocks distributed among the nodes is larger than 600 kB. 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. We show worst case analyses proving this fact, concluding that LZW-RLRU compression is the most suitable in this context since it is the most efficient one.

A. Worst Case for the Standard Distributed Implementation

In [2], the notions of bounded memory on-line decodable optimal LZW compression for the FREEZE and RESTART heuristics were introduced.

A *feasible* d-frozen LZW factorization $S = f_1 \cdots f_k$ is a feasible LZW factorization, where the number of different concatenations of a factor with the next character is $\leq d$. We define *optimal* d-frozen LZW factorization to be the feasible d-frozen LZW factorization with the snallest number of factors. Computing the optimal solution in polynomial time is quite straightforward if the degree of the polynomial time function is the dictionary size but it is obviously unpractical and a better algorithm is not known.

A *feasible* d-restarted LZW factorization $S = f_1 \cdots f_j \cdots f_i \cdots f_k$ is a feasible LZW factorization such that if j and i are consecutive indices where the restart operation happens, then the number of different concatenations of a factor with the next character is $\leq d$ between f_j and f_i. We define *optimal* d-restarted LZW factorization to be the feasible d-restarted LZW factorization with the smallest number of factors. A practical algorithm to compute the optimal solution is obviously not known as for the optimal d-frozen LZW factorization.

The compression models just introduced employ dictionaries with size bounded by the FREEZE and RESTART heuristics, respectively. The on-line greedy factorizations are obviously feasible. Moreover, feasible factorizations are the ones produced by the distributed algorithms described in the previous section. In this section, we give upper bounds to the approximation multiplicative factor. A trivial upper bound to the approximation multiplicative factor of every feasible factorization with respect to the optimal one is the maximum factor length of the optimal string factorization, that is, the height of the trie storing the dictionary. Such upper bound is $\Theta(d)$, where d is the dictionary size ($O(d)$ follows from the feasibility of the factorization and $\Omega(d)$ from the factorization of the unary string). There are strings for which the on-line greedy d-frozen LZW factorization is a $\Theta(d)$ approximation of the optimal one. Indeed, if we bound the dictionary size to $d + 2$ and consider the input binary string $(\prod_{i=0}^{d/2-1} ab^i ba^i)(\prod_{i=1}^{d} a^{d/2})$ then the on-line greedy *d-frozen* LZW factorization is a, b, ab, ba, abb, baa, $\cdots$ ab^i, ba^i, $\cdots$ $ab^{d/2-1}$, $ba^{d/2-1}$, a, a, $\cdots$, a while the optimal *d-frozen* LZW factorization is a, b, ab, b, a, abb, b, aa, $\cdots$ ab^i, b, a^i, $\cdots$ $ab^{d/2-1}$, b, $a^{d/2-1}$, $a^{d/2}$, $a^{d/2}$, $\cdots$ $a^{d/2}$. It

follows that the cost of the greedy factorization is $d + d^2/2$ while the cost of the optimal one is $5d/2 - 1$.

The feasible d-restarted LZW factorizations output by the distributed algorithms of the previous section can be as bad as the greedy solution using the frozen dictionary in the worst case. Indeed, if we apply any of such distributed algorithms to the input block of length d^2

$$b^{d^2/4-d/2}(\prod_{i=0}^{d/2-1} ab^iba^i)(\prod_{i=1}^{d} a^{d/2})$$

the dictionary is filled up by the greedy factorization process applied to the first half of the block, that is, $b^{d^2/4-d/2}(\prod_{i=0}^{d/2-1} ab^iba^i)$. Such factorization is: b, bb, $\cdots$, b^ℓ, $b^{\ell'}$, a, b, ab, ba, abb, baa, $\cdots$ ab^i, ba^i, $\cdots$ $ab^{d/2-1}$, $ba^{d/2-1}$ where $\ell' \leq \ell+1$ and the dictionary size is $d+\ell+3$. The static factorization of the second half is $a, a, \cdots a, a$ and the total cost of the factorization of the block is $\ell+1+d+d^2/2$ which is $\Theta(d^2)$. On the other hand, the cost of the optimal solution on the block is $\ell + 5d/2$, which is $\Theta(d)$. Observe that the $O(d)$ approximation multiplicative factor depends on the static phase and this happens when the dictionary learned on the first half of the block performs badly on the second half, that is in practice, when the data are highly disseminated. We will show in the next subsection that the on-line greedy d-restarted LZW factorization performs much better in the worst case, suggesting a more robust approach to distributed computing.

B. Worst Case Analysis of the Sequential Implementation

During the learning process before freezing and eventually restarting the dictionary, the on-line greedy factorization is the only feasible factorization producing factors which are all different from each other, that is, the number of factors equals the number of dictionary elements. This is the property we use to prove our result.

Theorem. The on-line greedy d-restarted LZW factorization is an $O(\sqrt{d})$ approximation of the optimal one, where d is the dictionary size.

Proof. Without loss of generality, we can assume the restart operation happens as soon as the dictionary is filled up during the greedy factorization process, since the static phase monitors the performance of the procedure. Let S be a string of length n and T be the trie storing the dictionary of factors of the optimal d-restarted LZW factorization Φ of S between two consecutive positions, where the restart operation happens. Each dictionary element (but the alphabet characters) corresponds to the concatenation of a factor f of the optimal factorization with the first character of the next factor, that we call an *occurrence* of the dictionary element (node of the trie) in Φ. We call an element of the dictionary, built by the greedy process, *internal* if its occurrence is contained in the occurrence of a node of T and denote with M_T the number of internal occurences. The number of non-internal occurences is less than the number of factors of Φ. Therefore, we can consider only the internal ones. An occurrence f' of the greedy factorization internal to a factor f of Φ is represented by a subpath of the path representing f in T. Let u be the endpoint at the lower level in T of this subpath (which, obviously, represents a prefix of f). Let $d(u)$ be the number of subpaths representing internal phrases with endpoint u and let $c(u)$ be the total sum of their lengths. All the occurences of the greedy factorization are different from each other between two consecutive positions, where the restart operation of the greedy procedure happens. Since two subpaths with the same endpoint and equal length represent the same factor, we have $c(u) \geq d(u)(d(u)+1)/2$. Therefore

$$1/2 \sum_{u \in T} d(u)(d(u)+1) \leq \sum_{u \in T} c(u) \leq 2|\Phi|H_T$$

where H_T is the height of T, $|\Phi|$ is the number of phrases of Φ and the multiplicative factor 2 is due to the fact that occurrences of dictionary elements may overlap. We denote with $|T|$ the number of nodes in T; since $M_T = \sum_{u \in T} d(u)$, we have

$$M_T^2 \leq |T| \sum_{u \in T} d(u)^2 \leq |T| \sum_{u \in T} d(u)(d(u)+1) \leq 4|T||\Phi|H_T$$

where the first inequality follows from the fact that the arithmetic mean is less than the quadratic mean. Then

$$M_T \leq \sqrt{4|T||\Phi|H_T} = |\Phi|\sqrt{\frac{4|T|H_T}{|\Phi|}} \leq 2|\Phi|\sqrt{H_T}$$

The statement of the theorem follows from the fact that the height of the trie is $\Theta(d)$ in the worst case. q. e. d.

The theorem suggests an approach restarting the dicionary as soon as it is filled up, which is more robust but in some cases (when the data are quite homegeneous) a little less effective in terms of compression effectiveness. Therefore, on a distributed system each processor stores a block of data and applies the on-line greedy LZW factorization adding a new element to the dictionary at each step. Obviously, blocks are short enough to observe the dictionary size bound d. From the the statement of the theorem in the previous section, such approach outputs an $O(\sqrt{d})$ approximation of the optimal solution since it computes the on-line greedy d-restarted factorization. If the file size is very large and the bound to the dictionary size is reached by one processor before the end of its block, a "least recently used" strategy can be applied to remove dictionary elements to preserve

robustness. The relaxed version of LZW-LRU compression using only two eqivalence classes is the one we propose as the most suitable and efficient for large size files lossless compression.

C. LZW-RLRU2 Compression: A Robust Approach

The relaxed version of the LRU heuristic using p equivalence classes 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 $\mu_0\mu_1\rho_1\mu_2\rho_2$ implements it as the one for the LZSS compressor explained in Section VI. A sequence of the same length works symmetrically for decompression.

VIII. Conclusion

We showed how to implement Lempel-Ziv data compression in the MapReduce framework for Web computing. An alternative to standard versions of the Lempel-Ziv method is proposed as the most efficient one for large size files compression. The robustness of the approach is evidentiated by a theoretical worst case analysis of the standard techniques. Moreover, scalability is preserved 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 but 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 [14], 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 [8], which are suitable for arbitrary size highly disseminated files. As future work, it is worth investigating experimentally if any of these PRAM algorithms (which are completely different from the ones presented in this paper) can be realized with MapReduce in practice on specific files.

References

[1] S. De Agostino, "Compressing Large Size Files on the Web in MapReduce," Proceedings International Conference on Internet and Web Applications and Services (ICIW), 2013, pp. 135-140.

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

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

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

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

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

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

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

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

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

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

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

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

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

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

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

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

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

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

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

[26] R. P. Brent, "A Linear Algorithm for Data Compression," Australian Computer Journal, vol. 19, 1987, pp. 64-68.

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

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

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

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

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

[32] 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, 1995, pp. 172-181.