The Greedy Approach to Dictionary-Based Static Text Compression on a Distributed System

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

***Abstract*—The greedy approach to dictionary-based static text compression can be executed by a finite state machine. When it is applied in parallel to different blocks of data independently, there is no lack of robustness even on standard large scale distributed systems with input files of arbitrary size. Beyond standard large scale, a negative effect on the compression effectiveness is caused by the very small size of the data blocks. A robust approach for extreme distributed systems is presented in this paper, where this problem is fixed by overlapping adjacent blocks and preprocessing the neighborhoods of the boundaries.**

Keywords-lossless compression; string factorization; parallel computing; distributed system; scalability; robustness

I. INTRODUCTION

Static data compression implies the knowledge of the input type. With text, dictionary based techniques are particularly efficient and employ string factorization. The dictionary comprises typical factors plus the alphabet characters in order to guarantee feasible factorizations for every string. Factors in the input string are substituted by pointers to dictionary copies and such pointers could be either variable or fixed length codewords.

The optimal factorization is the one providing the best compression, that is, the one minimizing the sum of the codeword lengths. Efficient sequential algorithms for computing optimal solutions were provided by means of dymamic programming techniques [1] or by reducing the problem to the one of finding a shortest path in a directed acyclic graph [2]. From the point of view of sequential computing, such algorithms have the limitation of using an off-line approach. However, decompression is still on-line and a very fast and simple real time decoder outputs the original string with no loss of information. Therefore, optimal solutions are practically acceptable for read-only memory files where compression is executed only once. Differently, simpler versions of dictionary based static techniques were proposed which achieve nearly optimal compression in practice.

An important simplification is to use a fixed length code for the pointers, so that the optimal decodable compression for this coding scheme is obtained by minimizing the number of factors. Such variable to fixed length approach is robust since the dictionary factors are typical patterns of the input specifically considered. The problem of minimizing the number of factors gains a relevant computational advantage by assuming that the dictionary is *prefix* (*suffix*), that is, all the prefixes (suffixes) of a dictionary element are dictionary elements [3]-[5]. The left to right greedy approach is optimal only with suffix dictionaries. An optimal factorization with prefix dictionaries can be computed on-line by using a semi-greedy procedure [4], [5]. On the other hand, prefix dictionaries are easier to build by standard adaptive heuristics [6], [7]. These heuristics are based on an "incremental" string factorization procedure [8], [9]. The most popular for prefix dictionaries is the one presented in [10]. However, the prefix and suffix properties force the dictionary to include many useless elements which increase the pointer size and slightly reduce the compression effectiveness. Moreover, the greedy approach to dictionary-based static text compression is optimal, in practice, for any kind of dictionary even if the theoretical worst case analysis shows that the multiplicative approximation factor with respect to optimal compression achieves the maximum length of a dictionary element. A more natural dictionary with no prefix and no suffix property is the one built by the heuristic in [11] or by means of separator characters as, for example, space, new line and punctuation characters for strings of a natural language. Finally, given an arbitrary dictionary, greedy static dictionary-based compression can be executed by a finite state machine.

Theoretical work was done, mostly in the nineties, to design efficient parallel algorithms on a random access parallel machine (PRAM) for dictionary-based static text compression [12]-[20]. Although the PRAM model is out of fashion today, shared memory parallel machines offer a good computational model for a first approach to parallelization. When we address the practical goal of designing distributed algorithms we have to consider 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

 ISBN: 978-1-61208-354-4

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. Parallel decompression is, obviously, possible on this model [15]. With parallel compression, the main issue is the one concerning scalability and robustness. Standardly, the scale of a system is considered large when the number of nodes has the order of magnitude of a thousand. Modern distributed systems may nowadays consist of hundreds of thousands of nodes, pushing scalability well beyond traditional scenarios (extreme distributed systems).

In [21], an approximation scheme of optimal compression with static prefix dictionaries was presented for massively parallel architectures, using no interprocessor communication during the computational phase since it is applied in parallel to different blocks of data independently. The scheme is algorithmically related to the semi-greedy approach previously mentioned and implementable on extreme distributed systems because adjacent blocks overlap and the neighborhoods of the boundaries are preprocessed. However, with standard large scale the overlapping of the blocks, the preprocessing of the boundaries and the prefix property of the dictionary are not necessary to achieve nearly optimal compression. Starting from this observation, we present in this paper two implementations of the greedy approach to static text compression with an arbitrary dictionary on a large scale and an extreme distributed system, respectively.

In Section II, we describe the different approaches to dictionary-based static text compression. The previous work on parallel approximations of optimal compression with prefix dictionaries is given in Section III. Section IV shows the two implementations of the greedy approach for arbitrary dictionaries. Conclusions and future work are given in Section V.

II. Dictionary-Based Static Text Compression

In this section, we describe the main dictionary-based static compression techniques and make a greedy versus optimal analysis. Then, we provide a finite state machine implementation of the greedy approach with an arbitrary dictionary.

A. Optimal Solutions

As mentioned in the introduction, the dictionary comprises typical factors (including the alphabet characters) associated with fixed or variable length codewords. The optimal factorization is the one minimizing the sum of the codeword lengths and sequential algorithms for computing optimal solutions were provided by means of dynamic programming techniques [1] or by reducing the problem to the one of finding a shortest path in a directed acyclic graph [2]. With suffix dictionaries we obtain optimality by means of a simple left to right greedy approach, that is, advancing with the on-line reading of the input string by selecting the longest matching factor with a dictionary element. Such procedure can be computed in real time by storing the dictionary in a *trie* data structure (a trie is a tree where the root represents the empty string and the edges are labeled by the alphabet characters). If the dictionary is prefix and the codewords length is fixed, there is an optimal semi-greedy factorization which is computed by the procedure of Fig. 1 [4], [5]. 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. The real time implementation employs an augmented trie data structure, obtained by modifying the original one [5].

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.

The semi-greedy factorization can be generalized to any dictionary by considering only those positions, among the ones covered by the current factor, next to a prefix that is a dictionary element [4]. We will see in the next subsection that the generalized semi-greedy factorization procedure is not optimal while the greedy one is not optimal even when the dictionary is prefix.

B. Greedy versus Optimal Factorizations

The maximum length of a dictionary element is an obvious upper bound to the multiplicative approximation factor of any string factorization procedure with respect to the optimal solution. We show that this upper bound is tight for the greedy and semi-greedy procedures when the dictionary is arbitrary. Such tightness is kept by the greedy procedure even if the dictionary is prefix. Let $baba^n$ be the input string and let $\{a, b, bab, ba^n\}$ be the dictionary. Then, the optimal factorization is b, a, ba^n while $bab, a, a, ..., a, ...a$ is the factorization obtained whether the greedy or the semi-greedy procedure is applied. On the other hand, with the prefix dictionary $\{a, b, ba, bab, ba^k : 2 \leq k \leq n\}$, the optimal factorization ba, ba^n is computed by the semi-greedy approach while the greedy factorization remains the same. These examples, obviously, prove our statement on the tightness of the upper bound.

C. The Finite State Machine Implementation

We show the finite state machine implementation producing the on-line greedy factorization of a string with an arbitrary dictionary. The most general formulation for

a finite state machine M is to define it as a sixtuple $(A, B, Q, \delta, q_0, F)$ with an input alphabet A, an output alphabet B, a set of states Q, a transition function $\delta : QxA- > QxB^*$, an initial state q_0 and a set of accepting states $F \subseteq Q$. The trie storing the dictionary is a subgraph of the finite state machine diagram. It is well-known that each dictionary element is represented as a path from the root to a node of the trie where edges are labeled with an alphabet character (the root representing the empty string). The edges are directed from the parent to the child and the set of nodes represent the set of states of the machine. The output alphabet is binary and the factorization is represented by a binary string having the same length of the input string. The bits of the output string equal to 1 are those corresponding to the positions where the factors start. Since every string can be factorized, every state is accepting. The root represents the initial state. We need only to complete the function δ, by adding the the missing edges of the diagram. The empty string is associated as output to the edges in the trie. For each node, the outcoming edges represent a subset of the input alphabet. Let f be the string (or dictionary element) corresponding to the node v in the trie and a an alphabet character not represented by an edge outcoming from v. Let $fa = f_1 \cdots f_k$ be the on-line greedy factorization of fa and i the smallest index such that $f_{i+1} \cdots f_k$ is represented by a node w in the trie. Then, we add to the trie a directed edge from v to w with label a. The output associated with the edge is the binary string representing the sequence of factors $f_1 \cdots f_i$. By adding such edges, the machine is entirely defined. Redefining the machine to produce the compressed form of the string is straightforward.

III. Previous Work

Given an arbitrary dictionary, for every integer k greater than 1 there is an $O(km)$ time, $O(n/km)$ processors distributed algorithm factorizing an input string S with a cost which approximates the cost of the optimal factorization within the multiplicative factor $(k+m-1)/k$, where n and m are the lengths of the input string and the longest factor respectively [12]. However, with prefix dictionaries a better approximation scheme was presented in [21], producing a factorization of S with a cost approximating the cost of the optimal factorization within the multiplicative factor $(k+1)/k$ in $O(km)$ time with $O(n/km)$ processors. This second approach was designed for massively parallel architecture and is suitable for extreme distributed systems, when the scale is beyond standard large values. On the other hand, the first approach applies to standard small, medium and large scale systems. Both approaches provide approximation schemes for the corresponding factorization problems since the multiplictive approximation factors converge to 1 when km converge to n. Indeed, in both cases compression is applied in parallel to different blocks of data independently. Beyond standard large scale, adjacent blocks overlap and the neighborhoods of the boundaries are preprocessed.

To decode the compressed files on a distributed system, it is enough to use a special mark occurring in the sequence of pointers each time the coding of a block ends. The input phase distributes the subsequences of pointers coding each block among the processors. Since a copy of the dictionary is stored in every processor, the decoding of the blocks is straightforward.

In the following two subsections, we describe the two approaches. Then, how to speed up the preprocessing phase of the second approach is described in the last subsection. In the next section, we present new results by arguing that we can relax on the requirement of computing a theoretical approximation of optimal compression since, in practice, the greedy approach is optimal on data blocks sufficiently long. On the other hand, when the blocks are too short since the scale of the distributed system is beyond standard values, the overlapping of the adjacent blocks and the preprocessing of the neighborhoods of the boundaries are sufficient to garantee the robustness of the greedy approach.

A. Standard Scale Distributed Systems

Given an input string of length n, we simply apply in parallel optimal compression to blocks of length km, with k integer greater than one and m maximum length of a factor as stated at the beginning of this section. Every processor stores a copy of the dictionary. For arbitrary dictionary, we execute the dynamic programming procedure computing the optimal factorization of a string in linear time [1] (the procedure in [2] is pseudo-linear for fixed-length coding and, even, super linear for variable length). Obviously, this works for prefix and suffix dictionaries as well and, in any case, we know the semi-greedy and greedy approach are implementable in linear time. It follows that the algorithm requires $O(km)$ time with n/km processors and the multiplicative approximation factor is $(k+m-1))/k$ with respect to any factorization. Indeed, when the boundary cuts a factor the suffix starting the block and its substrings might not be in the dictionary. Therefore, the multiplicative approximation factor follows from the fact that $m-1$ is the maximum length for a proper suffix as shown in Fig. 2 (sequence of plus signs in parentheses). If the dictionary is suffix, the multiplicative approximation factor is $(k+1))/k$ since each suffix of a factor is a factor.

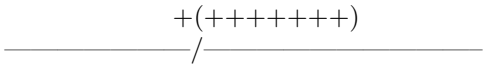

Figure 2. The making of the surplus factors.

The approximation scheme is suitable only for standard

scale systems unless the file size is very large. In effect, the block size must be the order of kilobytes to guarantee robustness. Beyond standard large scale, overlapping of adjacent blocks and a preprocessing of the boundaries is required as we will see in the next subsection.

B. Beyond Standard Large Scale

With prefix dictionaries a better approximation scheme was presented in [21]. During the input phase blocks of length $m(k+2)$, except for the first one and the last one which are $m(k+1)$ long, are broadcasted to the processors. Each block overlaps on m characters with the adjacent 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).

We call a *boundary match* a factor covering positions in the first and second half of the $2m$ characters shared by two adjacent blocks. The processors execute the following algorithm to compress each block:

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

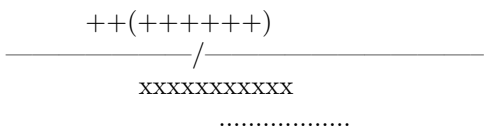

Figure 3. The making of a surplus factor.

Stopping the factorization of each 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 other factorization. Indeed, as it is shown in Fig. 3, the factor in front of the right boundary match (sequence of x's) might be extended to be a boundary match itself (sequence of plus signs) and to cover the first position of the factor after the boundary (dotted line). Then, the approximation scheme produces a factorization of S with a cost approximating the cost of the optimal factorization within the multiplicative factor $(k+1)/k$ in $O(km)$ time with $O(n/km)$ processors.

In [21], it is shown experimentally that for $k = 10$ the compression ratio achieved by such factorizarion is about the same as the sequential one and, consequently, the approach is suitable for extreme distributed systems, as we will explain in the next section.

C. Speeding up the Preprocessing

The parallel running time of the preprocessing phase computing the boundary matches is $O(m^2)$ by brute force. To lower the complexity to $O(m)$, an augmented trie data structure is needed. For each node v of the trie, let f be the dictionary element corresponding to v and a an alphabet character not represented by an edge outcoming from v. Then, we add an edge from v to w with label a, where w represents the longest proper suffix of fa in the dictionary. Each processor has a copy of this augmented trie data structure and first preprocess the $2m$ characters overlapped by the adjacent block on the left boundary and, secondly, the ones on the right boundary. In each of these two sub-phases, the processors advance with the reading of the $2m$ characters from left to right, starting from the first one while visiting the trie starting from the root and using the corresponding edges. A temporary variable t_2 stores the position of the current character during the preprocessing while another temporary variable t_1 is, initially, equal to t_2. When an added edge of the augmented structure is visited, the value $t = t_2 - d + 1$ is computed where d is the depth of the node reached by such edge. If t is a position in the first half of the $2m$ characters, then t_1 is updated by changing its value to t. Else, the procedure stops and t_2 is decreased by 1. If t_2 is a position in the second half of the $2m$ characters then t_1 and t_2 are the first and last position of a boundary match, else there is no boundary match.

IV. The Greedy Approach

In practice, greedy factorization is nearly optimal. As a first approach, we simply apply in parallel left to right greedy compression to blocks of length km. With standard scale systems, the block size must be the order of kilobytes to guarantee robustness. Each of the $O(n/km)$ processors could apply the finite state machine implementation of subsection II.C to its block.

Beyond standard large scale, overlapping of adjacent blocks and a preprocessing of the boundaries are required as for the optimal case. Again, during the input phase overlapping blocks of length $m(k+2)$ are broadcasted to the processors as in the previous section. On the other hand, the definition of boundary match is extended to those factors, which are suffixes of the first half of the $2m$ characters shared by two adjacent blocks. The following procedure, even if it is not an approximation scheme from a theoretical point of view, performs similarly (observe that, in this case, we compute the longest boundary match rather than the one ending furthest to the right):

- For each block, every corresponding processor but the one associated with the last block computes the longest boundary match between its block and the next

one;

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

To lower the parallel running time of the preprocessing phase to O(m), the same augmented trie data structure, described in the previous section, is needed but, in this case, the boundary matches are the longest ones rather than the ones ending furthest to the right. Then, besides the temporary variables t_1 and t_2, employed by the preprocessing phase described in the previous section, two more variables τ_1 and τ_2 are required and, initially, equal to t_1 and t_2. Each time t_1 must be updated by such preprocessing phase, before it the value $t_2 - t_1 + 1$ is compared with $\tau_2 - \tau_1$. If it is greater or τ_2 is smaller than the last position of the first half of the $2m$ characters, τ_1 and τ_2 are set equal to t_1 and $t_2 - 1$. Then, t_1 is updated. At the end of the procedure, τ_1 and τ_2 are the first and last positions of the longest boundary match. We wish to point out that there is always a boundary match that is computed, since the final value of τ_2 always corresponds to a position equal either to one in the second half of the $2m$ characters or to the last position of the first half. Again, after preprocessing each of the O(n/km) processors could apply the finite state machine implementation of subsection II.C to its block.

The approach is nearly optimal for k = 10, as the approximation scheme of previous section. The compression ratio achieved by such factorizarion is about the same as the sequential one. Considering that typically the average match length is 10, one processor can compress down to 100 bytes independently. This is why the approximation scheme was presented for massively parallel architecture and the approach, presented in this section, is suitable for extreme distributed systems, when the scale is beyond standard large values. Indeed, with a file size of several megabytes or more, the system scale has a greater order of magnitude than the standard large scale parameter. 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 1K. With standard large scale systems the block length is several kilobytes with just a few megabytes to compress and the approach using boundary matches is too conservative.

V. Conclusion

We presented parallel implementations of the greedy approach to dictionary-based static text compression suitable for standard and non-standard large scale distributed systems. In order to push scalability beyond what is traditionally considered a large scale system, a more involved approach distributes overlapping blocks to compute boundary matches. These boundary matches are relevant to maintain the compression effectiveness on a so-called extreme distributed system. If we have a standard small, medium or large scale system available, the approach with no boundary matches can be used. The absence of a communication cost during the computation guarantees a linear speed-up. Moreover, the finite state machine implementation speeds up the execution of the distributed algorithm in a relevant way when the data blocks are large, that is, when the size of the input file is large and the size of the distributed system is relatively small. As future work, experiments on parallel running times should be done to see how the preprocessing phase effects on the linear speed-up when the system is scaled up beyond the standard size and how relevant the employment of the finite state machine implementation is when the data blocks are very small.

References

[1] R. A. Wagner, "Common Phrases and Minimum Text Storage," Communications of the ACM, vol. 16, 1973, pp. 148-152.

[2] E. J. Shoegraf and H. S. Heaps, "A Comparison of Algorithms for Data Base Compression by Use of Fragments as Language Elements," Information Storage and Retrieval, vol. 10, 1974, pp. 309-319.

[3] M. Cohn and R. Khazan, "Parsing with Suffix and Prefix Dictionaries," Proceedings IEEE Data Compression Conference, 1996, pp. 180-189.

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

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

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

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

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

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

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

[11] V. S. Miller and M. N. Wegman, "Variations on Theme by Ziv - Lempel," Combinatorial Algorithms on Words (eds. Apostolico, A., Galil, Z.), Springer, 1985, pp. 131-140

 ISBN: 978-1-61208-354-4

[12] 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.

[13] S. De Agostino, Sub-Linear Algorithms and Complexity Issues for Lossless Data Compression, Master's Thesis, Brandeis University, 1994.

[14] S. De Agostino, Parallelism and Data Compression via Textual Substitution, Ph. D. Dissertation, Sapienza University of Rome, 1995.

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

[16] S. De Agostino S. and J. A. Storer, "Parallel Algorithms for Optimal Compression Using Dictionaries with the Prefix Property," Proceedings IEEE Data Compression Conference, 1992, pp. 52-61.

[17] D. S. Hirschberg and L. M. Stauffer, "Parsing Algorithms for Dictionary Compression on the PRAM," Proceedings IEEE Data Compression Conference, 1994, pp. 136-145.

[18] D. S. Hirschberg and L. M. Stauffer, "Dictionary Compression on the PRAM," Parallel Processing Letters, vol. 7, 1997, pp. 297-308.

[19] H. Nagumo, M. Lu and K. Watson, "Parallel Algorithms for the Static Dictionary Compression," Proceedings IEEE Data Compression Conference, 1995, pp. 162-171.

[20] L. M. Stauffer and D. S. Hirschberg, "PRAM Algorithms for Static Dictionary Compression," Proceedings International Symposium on Parallel Processing, 1994, pp. 344-348.

[21] 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.

 ISBN: 978-1-61208-354-4