Compressing Distributed Text in Parallel with (s, c)-Dense Codes

Carolina Bonacic[1] Antonio Fariña[3] Mauricio Marín[4] Nieves R. Brisaboa[3]

[1]Departamento Cs. Computación, Universidad de Chile
[3]Laboratorio de Bases de Datos, Facultade de Informática, Universidad da Coruña, España
[4]Departamento de Computación, Universidad de Magallanes, Chile

Contact Author's E-Mail: Mauricio.Marin@umag.cl

Abstract

Systems able to cope with very large text collections are making intensive use of distributed memory parallel computing platforms such as Clusters of PCs. This is particularly evident in Web Search Engines which must resort to parallelism in order to deal efficiently with both high rates of queries per unit time and high space requirements in the form of large numbers of small documents stored in secondary memory. Those documents can be stored in compressed format to reduce memory space and communication time. This paper proposes a parallel algorithm for compressing text in such a distributed memory environment. We show efficient performance against the usual-practice alternative of compressing the whole text on a single machine.

1 Introduction

Over the last decade it has become relevant to study algorithms devised to reduce the amount of space occupied by very large text databases (giga/tera bytes). In addition, it has become clear the convenience of letting query operations be performed directly on the compressed text to avoid decompression on the fly. On the other hand, it has also become evident that parallel computing can be an effective tool for reducing running times in systems that demand the processing of large numbers of queries on text databases. Here we can also profit from reduced communication as a result of transmitting compressed text/queries.

Compression algorithms save space by replacing the most frequent words in the text with symbols (codes) requiring much less space (a few bits). Notice that the compression process can take a significant amount of running time as this process at least grows up linearly with the size n of the text database. On a p processors parallel computer we should be able to improve this to $O(n/p)$.

Modern and truly scalable parallel computer architectures are those based on the distributed-memory sharing-nothing model. In this case, the practical alternative is just distributing evenly

the text on the different processors and a straightforward, but naive, approach in this case would be to compress co-resident text using a standard sequential algorithm in every processor.

However, this would involve dealing with different word coding in every processor as a result of considering only a subset of the whole text database. Queries would have to be transmitted to their target processors in its original form and once there they would have to be compressed to proceed with the scanning of co-resident text. Then if the processing of a query would have to be continued in another processor it would be necessary a decompression before leaving the current processor. For hundreds of queries per second this approach is clearly not efficient. In addition, for systems in which new text is constantly added to and old one is removed from (e.g., a news service) it is desirable to have a fast method for maintaining the global consistency.

This paper presents a parallel compression algorithm which works on a text database distributed on p processors. We propose the parallelization of the (s, c) Dense Code compression algorithm presented in [3], which is a generalization of the algorithm presented in [2]. This parallelization is effected on top of the BSP model of parallel computing [8, 12]. The model allows the design of software that is portable and we show that the proposed algorithm is scalable and efficient in practice. BSP algorithms can be run without changes on differing parallel architectures ranging from super-computers to clusters of PCs. A convenient feature of BSP is that its well-defined structure enables performance prediction in a similar way to sequential algorithm design. We show that the proposed algorithm is able to compress distributed text in $O(n/p)$ running time and its design is such that it let processors work in an almost perfect balanced manner at reduced communication and synchronization costs.

Classic compression techniques such as the well-known algorithms of Ziv and Lempel [13, 14] or the character oriented code of Huffman [5], are not suitable for large textual databases. An important disadvantage of these techniques is the inefficiency of searching for words directly on the compressed text. Compression schemes based on Huffman codes are not often used on natural language text because of the poor compression ratios achieved. On the other hand, Ziv and Lempel algorithms obtain better compression ratios, but the search for a word on the compressed text is inefficient [7].

One of the compression schemes presented in [4] is based on plain Huffman coding over words and allows the fast search for a word on the compressed text without decompressing it. Here text words are considered the symbols to be compressed and thereby the usual searching and indexing techniques can work on the compressed representation of words [6, 15]. In [2] an improved scheme is presented which achieves better compression ratios, allows searching directly on the compressed text, has a simpler and smaller vocabulary representation and has a simpler and faster coding strategy. The work in [3] further improves compression ratios by adjusting codes generation to the particular word distribution in the given text database.

The remaining of this paper is organized as follows. Section 2 presents a description of the model of parallel computing. Section 3 describes the sequential compression algorithm and section 4 presents its parallelization and analysis. Section 5 presents conclusions.

2 The BSP model of Parallel Computation

The parallel compression algorithm we propose is devised upon the bulk-synchronous model of parallel computing (BSP model) [8, 12]. This is a distributed memory model with a well-defined structure that enables the prediction of running time. We use this last feature to compare different alternatives by considering their respective effects in communication and synchronization of processors. The model of computation ensures portability at the very fundamental level by allowing algorithm design in a manner that is independent of the architecture of the parallel computer. Shared and distributed memory parallel computers are programmed in the same way. They are considered emulators of the more general bulk-synchronous parallel machine.

In the BSP model of computing [8, 12] both computation and communication take place in bulk before the next point of global synchronization of processors. A BSP program is composed of a sequence of *supersteps*. During each superstep, the processors may only perform computations on data held in their local memories and/or send messages to other processors. These messages are available for processing at their destinations by the next superstep, and each superstep is ended with the barrier synchronization of the processors.

The practical model of programming is SPMD, which is realized as p program copies running on the p processors, wherein communication and synchronization among copies is performed by ways of libraries such as BSPlib [10] or BSPub [11]. Note that BSP is actually a paradigm of parallel programming and not a particular communication library. In practice, it is certainly possible to implement BSP programs using the traditional PVM and MPI libraries. A number of studies have shown that bulk-synchronous parallel algorithms lead to more efficient performance than their message-passing or shared-memory counterparts in many applications [8, 9].

The total running time cost of a BSP program is the cumulative sum of the costs of its supersteps, and the cost of each superstep is the sum of three quantities: w, $h\,g$ and l, where w is the maximum of the computations performed by each processor, h is the maximum of the messages sent/received by each processor with each word costing g units of running time, and l is the cost of barrier synchronizing the processors. The effect of the computer architecture is included by the parameters g and l, which are increasing functions of p. These values along with the processors speed s (e.g. mflops) can be empirically determined for each parallel computer by executing benchmark programs at installation [8].

3 The (s, c)-Dense Coding Compression Algorithm

Codes representing text elements are formed by sequences of base-c digits terminated by a digit between c and $c + s - 1$. Digits between 0 and $c - 1$ are called continuers whereas digits between c and $c + s - 1$ are called stoppers. Digits can use any number of bits within bytes and codes are stored using a certain number k of bytes. Thus there is a pair (c, s) for which it is possible to get the optimal compression. This depends on the particular distribution of words (elements) in the text. Correspondence between elements and codes is assigned in a sequential way by following a

decreasing frequency order. Let $k \geq 1$ be the number of bytes in each code, then k satisfies

$$s\frac{c^{k-1}-1}{c-1} \leq i < s\frac{c^k-1}{c-1}\,.$$

The code corresponding to text element i is formed by $k-1$ base-c digits plus a final digit. If $k=1$, the code is the stopper $c+i$ where $i = 0, 1, 2, ...$ indicate elements in decreasing frequency order with $i = 0$ being the most frequent one in the whole text. Elements $i > s$ requires one or more continuers. In that case the code is formed by the number $\lfloor x/s \rfloor$ written as a sequence of digits in base-c and ended by $c + (x \bmod s)$ with $x = i - \frac{sc^{k-1}-s}{c-1}$.

The optimal value for (s, c) is found by considering that $s + c = 2^b$ and $1 \leq s \leq 2^b - 1$ where in practice $b = 8$ bits. The first s elements in frequency decreasing order are encoded with 1 byte, the following $s\,c$ elements with 2 bytes, $s\,c^2$ with 3 bytes, $s\,c^3$ with 4 bytes and so on. As frequency information is determined for every text element it is possible to predict the final size of the compressed file. Then it suffices to test all possible values of s and choose the one which produces the minimal file size. Calculation of the compressed file size for a given s can be reduced by looking directly at the points in which coding passes from k to $k+1$ bytes and working with the cumulative frequency at those points. This is effected by advancing at $s\,c^k$ step increments and adding the difference between the cumulative frequency between the two inflection points.

The main steps followed by the sequential compression algorithm are the following (further details in [3]):

1. Assume a text database composed of n_e elements (words, symbols, spaces, etc.). A scan over the text determines a table T of n_v pairs (element, frequency) where frequency is the number of times the respective element appears in the text. This operation takes $O(n_e)$ running time.

2. The table T is sorted by decreasing frequency and then the cumulative frequency is calculated following the pairs from the largest frequency to the smallest one. The cost of these operations is $O(n_v \log n_v)$ units of running time.

3. Compute the value of s that produces the least compressed file size. This is effected by testing s values between 1 and $2^b - 1$ in a binary search fashion. For each value of s the determination of the file size can be made in $O(\log n_v \log\log n_v)$ time [3].

4. Build up a codes table C with pairs (element, code) by following the frequency table T in decreasing frequency order and assigning codes from one byte first to two or more bytes last. This has cost $O(n_v)$.

5. Produce the compressed file by transforming every element in the input file to its corresponding code in table C and storing it in the output file. This takes $O(n_e)$ time.

Thus the cost of the sequential compression algorithm is mainly $O(n_e)$. This because the size n_v of the vocabulary table T grows very slowly when the text size n_e is very large (it has been conjectured that it achieves a constant value for huge n_e [1]).

4 Parallel Compression with (s, c)-Dense Coding

We assume that the text is evenly distributed on p processors such that co-resident text has size n_e/p. The algorithm proposed in this paper compress the whole text in $O(n_e/p)$ computation time with very low overheads in communication and synchronization. The description is based on the 5 steps of the above described sequential algorithm.

The proposed BSP algorithm takes 6 supersteps to complete the compression work. Every processor i effects the following operations,

Superstep 1

- Every processor i builds table T_i of size n_v at cost $n_e/p + n_v$.
- Each pair (e, q) of table T_i is sent to processor $j = \text{hash}(e)$ where the hash function distributes evenly elements e on the p processors. This operation costs $n_v\, g + l$.

Superstep 2

- Get the n_v arriving pairs (e_i, q_i), (e_j, q_j), (e_k, q_k), ..., and calculate $(e_i, q_i + q_j + q_k + ...)$ for all cases in which $e_i = e_j = e_k = ...$ At this point in every pair (e, q) the value of q is the total number of occurrences (frequency) of element e in the whole text database. In every processor i those values are stored in table T_i^* of size n_v/p. The cost of this operation is n_v.

BSP–Sort(T_i^*)

- The p tables T_i^* are sorted by decreasing frequency order using a standard BSP sort algorithm at cost $O((n_v/p)\log(n_v) + (n_v/p)\, g + l)$. Let T_i^+ be the p sorted sets $0 \leq i \leq p-1$, each located in a different processor such that T_0^+ on processor 0 has the n_v/p elements with the largest frequency, T_1^+ on processor 1 has the n_v/p second ones, and so on.
- Every processor i adds the frequencies located in their respective tables T_i^+ and send the result to processors $i+1$, $i+2$, ..., $p-1$ at cost $O(n_v/p + p\, g)$.

Superstep 3

- Every processor i calculates the cumulative frequency by considering the sums received from processors 0, 1, ..., $i-1$ and the frequency values stored in their respective tables T_i^+ at cost $O(n_v/p)$. The cumulative frequencies are associated with the elements of T from the most frequent one to the least one.
- Calculate $2^b \log n_v$ indexes x related to the positions in the global table $T = \cup_{0 \leq i < p} T_i^+$ for the range of values of s and c with $c = 2^b - s$ and $1 \leq s \leq 2^b - 2$, such that the x values are the positions at which each x is increased as $x + s\, c^k$ with $k \geq 0$ and $0 \leq x \leq n_v$. This is effected to let every processor i work on a different range of $2^b/p$ values of s in order to determine the pair (s, c) that produces the optimal compressed file size. These calculations use a determined number of cumulative frequencies stored at positions given by the x values.

- Every processor i sends pairs $(x,\ u)$, where position x is in T_i^+ and u is its cumulative frequency, to the processor in charge of the respective range of values of s. The cost of this operation is $O(\ (2^b/p\ \log n_v\ )g\ )$.

Superstep 4

- Every processor i receives the pairs (x, u) and calculates the local minimum value of s and the respective file size using the sequential algorithm at cost $O(2^b/p\ \log n_v)$.
- Send the local minimum (s,size) to all other processors at cost $O(p\,g)$.

Superstep 5

- From the arriving local minima determine the optimal (s, c).
- Using the optimal (s, c) and information about how the global indexes to table $T = \cup_{0\leq i<p} T_i^+$ are distributed on the p processors, calculate the codes table C_i. This costs $O(n_v)$.
- Every processor i sends its codes table C_i to all others at cost $n_v\,g$.

Superstep 6

- Merge the arriving codes tables C_k at cost n_v.
- Compress the local text using table $C = \cup_{0\leq i<p} C_i$ at cost $O(n_e/p)$.

Then the total asymptotic cost of the compression BSP algorithm is given by

$$n_e/p + n_v + n_v\,g + l$$

Note that any parallel algorithm has to pay the communication cost $n_v\,g$ since the text is distributed on the p machines and all of them need the same codes table C of size n_v. If this table were calculated in just one machine this cost could increase to $n_v\,p\,g$.

Empirical Results

Experiments where performed to evaluate the practical performance of the proposed BSP algorithm. We worked on a number of PCs connected by a communication 100MB switch. Text come from a Chilean newspaper, Latex files and postscript files. Since we observed that the cost of communication was smaller than the cost of memory secondary, we performed experiments with fairly small texts since, for our PC cluster, those are the conditions in which the parallel algorithm is less efficient when compared to the sequential one. We worked with files sizes ranging from 1MB to 32MB.

We performed experiments to investigate the running time cost of the five main steps followed by the sequential algorithm. The first and last steps of the sequential algorithm take $O(n_e)$ time and our parallel algorithm can perform those in $O(n_e/p)$ time. Thus it is relevant to investigate the relative importance of the steps 2-4 so we can have a picture of how hard is to amortize communication and synchronization with the gain $O(n_e/p)$. The parallel algorithm can also amortize communication by performing the 2-4 steps over a n/p fraction of the data size. In all cases we

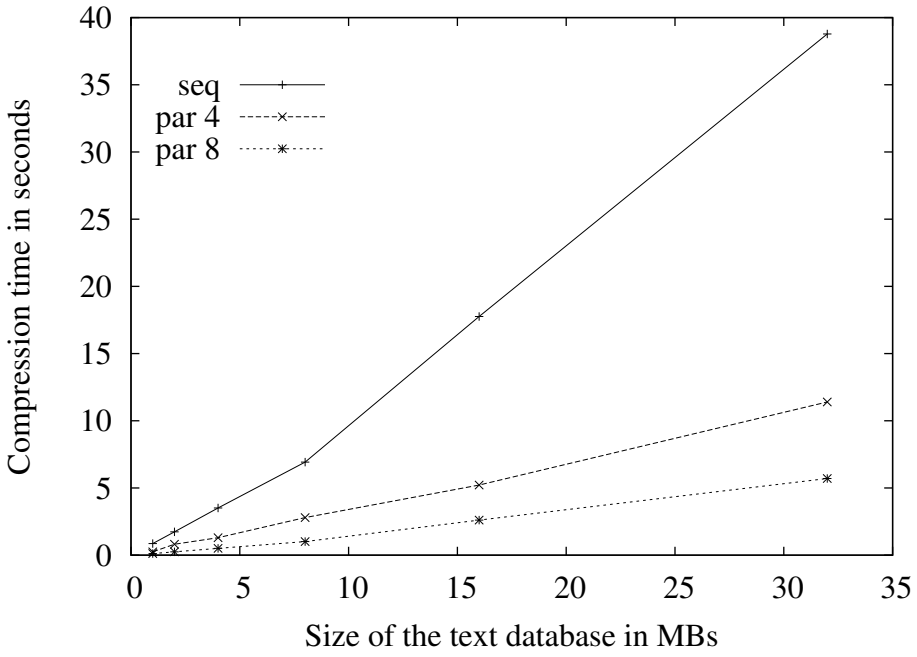

Figure 1: Speedups for 4 and 8 processors for text from a newspaper.

observed that the running time cost of steps 2-4 was below 15%. The cost of steps 1 and 5 was strongly influenced (70%) by the data structure used to handle tables T and C.

In figure 1 we show speedups for 4 and 8 processors on 1MB–32MB samples of a Chilean newspaper. Other format files performed in a similar way. It can be observed that the parallel algorithm can easily outperform the sequential one. This is mainly due to the $O(n_e/p)$ improvement on the steps 1 and 5 of the sequential algorithm. Note, however, that the sequential running times are quite high when compared with fast compressors such as the `GNU gzip` program. Our implementations are in C++ and we compiled with `GNU g++` with `-O3` compiler option. We have realized that we need to further improve our implementation of tables T and C as they are still an important fraction of the total running time.

Nevertheless the BSP cost of the algorithm we propose in this paper shows that it is indeed a scalable algorithm which can deal efficiently with very large text databases. This because the term n_v grows very slowly with n_e making that the dominant cost be the supersteps 1 (first point) and 6 (last point).

5 Conclusions

We have proposed a BSP realization of a recently developed compression algorithm [3]. The algorithm is efficient in practice because it is able to reduce the amount of communication and synchronization. Its asymptotic cost is $n_e/p + n_v + n_v\, g + l$ with low constant factors. For very large text databases (terabytes) the dominating factor is n_e/P since n_v tends to a constant or grows

very slowly.

The proposed BSP algorithm admits some low cost variants that simply its implementation. A crude but effective strategy can be to locally sort the resulting tables T_i^* in superstep 2 and immediately send those tables to all other processors. This costs $O(n_v/p \log n_v/p + n_v\, g)$. In superstep 3 we can merge the arriving tables so we end up with a vocabulary table that contains all text elements (words, symbols, etc) together with their global frequencies of occurrences in the whole text database. From this point onwards we can proceed as in the sequential algorithm. We can also include a couple of supersteps to let every processor work on the calculation of the optimal (s, c) in parallel and over a subset $2^b/p$ of the range of values of s. Of course, all of this performs more sequential work than the proposed BSP algorithm because no calculations over a subset n_v/p of the vocabulary are performed. However, we have observed that it has reasonable performance in practice.

We are currently improving the efficiency of the algorithms in charge of handling the vocabulary table T and the codes table C. Even though in our implementation we have them as hashing tables, they are still dominating the running time cost. Thus some extra care must be taken in order to make them faster so that our sequential realization can achieve running times fairly similar to standard compression programs.

References

[1] R. Baeza and B. Ribeiro. *Modern Information Retrieval*. Addison-Wesley., 1999.

[2] N. Brisaboa, E. Iglesias, G. Navarro, and J. Paramá. An Efficient Compression Code for Text Databases. In *Proceedings of the 25th European Conference on Information Retrieval Research (ECIR'03)*, LNCS 2633, pages 468–481, 2003.

[3] N. Brisaboa, A. Fari na, G. Navarro, and M. Esteller. (S,C)-Dense Coding: An Optimized Compression Code for Natural Language Text Databases. In *Proceedings of the 10th International Symposium on String Processing and Information Retrieval (SPIRE 2003)*, LNCS 2857, pages 122–136. Springer, 2003.

[4] E. Silva de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, April 2000.

[5] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. Inst. Radio Eng.*, 40(9):1098–1101, 1952.

[6] G. Navarro, E. Silva de Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3(1):49–77, 2000.

[7] G. Navarro and J. Tarhio. Boyer-Moore string matching over Ziv-Lempel compressed text. In *11th Annual Symposium on Combinatorial Pattern Matching (CPM'2000)*, pages 166–180. LNCS 1848, 2000.

[8] D.B. Skillicorn, J.M.D. Hill, and W.F. McColl. Questions and answers about BSP. Technical Report PRG-TR-15-96, Computing Laboratory, Oxford University, 1996. Also in *Journal of Scientific Programming*, V.6 N.3, 1997.

[9] D.B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 20(2):123–169, June 1998.

[10] URL. BSP and Worldwide Standard, http://www.bsp-worldwide.org/.

[11] URL. BSP PUB Library at Paderborn University, http://www.uni-paderborn.de/bsp.

[12] L.G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33:103–111, Aug. 1990.

[13] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[14] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

[15] N. Ziviani, E. Silva de Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next-generation of text retrieval systems. *IEEE Computer*, 33(11):37–44, 2000.