Parallel Lempel Ziv Coding

Shmuel Tomi Klein and Yair Wiseman

Computer Science Department, Bar-Ilan University, Ramat-Gan 52900, ISRAEL
{tomi,wiseman}@cs.biu.ac.il

Abstract

We explore the possibility of using multiple processors to improve the encoding and decoding times of Lempel Ziv schemes. A new layout of the processors, based on a full binary tree, is suggested and it is shown how LZSS and LZW can be adapted to take advantage of such parallel architectures. The layout is then generalized to higher order trees. Experimental results show an improvement in compression over the standard method of parallelization and an improvement in time over the sequential method.

Key words: Data Compression, Lempel-Ziv algorithms, parallel algorithms

1 Introduction

Compression methods are often partitioned into static and dynamic methods. The *static* methods assume that the file to be compressed has been generated according to a certain model which is fixed in advance and known to both compressor and decompressor. The model could be based on the probability distribution of the different characters or more generally of certain variable length substrings that appear in the file, combined with a procedure to parse the file into a well determined sequence of such elements. The encoded file can then be obtained by applying some statistical encoding function, such as Huffman or arithmetic coding. Information about the model is either assumed to be known (such as the distribution of characters in English text), or may be gathered in a first pass over the file, so that the compression process may only be performed in a second pass.

Many popular compression methods, however, are *adaptive* in nature. The underlying model is not assumed to be known, but discovered during the sequential processing of the file. The encoding and decoding of the i-th element is based on the distribution of the $i-1$ preceding ones, so that compressor and decompressor can work in synchronization without requiring the transmittal of the model itself. Examples of adaptive methods are the Lempel-Ziv (LZ) methods and their variants, but there are also adaptive versions of Huffman and arithmetic coding.

We wish to explore the possibility of using multiple processors to improve the encoding and decoding times. In [7] this has been done for static Huffman coding, focusing in particular on the decoding process. The current work investigates how parallel processing could be made profitable for Lempel Ziv coding.

Previous work on parallelizing compression includes [1, 2, 3], which deal with LZ compression, [5], relating to Huffman and arithmetic coding, and [4]. A parallel method for the construction of Huffman trees can be found in [8]. Our work concentrates on LZ methods, in particular a variant of LZ77, [14], known as LZSS, and a variant of LZ78, [15], known as LZW. In LZSS, [10], the encoded file consists of a sequence of items each of which is either a single character, or a pointer of the form (*off, len*) which replaces a string of length *len* that appeared *off* characters earlier in the file. Decoding of such a file is thus a very simple procedure, but for the encoding there is a need to locate longest reoccurring strings, for which sophisticated data structures like hash tables or binary trees have been suggested. In LZW, [11], the encoded file consists of a sequence of pointers to a *dictionary*, each pointer replacing a string of the input file that appeared earlier and has been put into the dictionary. Encoder and decoder must therefore construct identical copies of the dictionary.

The basic idea of parallel coding is partitioning the input file of size N into n blocks of size N/n and assigning each block to one of the n available processors. For static methods the encoding is then straightforward, but for the decoding, it is the compressed file that is partitioned into equi-sized blocks, so there might be a problem of synchronization at the block boundaries. This problem may be overcome by inserting dummy bits to align the block boundaries with codeword boundaries, which causes a negligible overhead if the block size is large enough. Alternatively, in the case of static Huffman codes, one may exploit their tendency to resynchronize quickly after an error, to devise a parallel decoding procedure in which each processor decodes one block, but is allowed to overflow into one or more following blocks until synchronization is reached, [7].

For dynamic methods one is faced with the additional problem that the encoding and decoding of elements in the i-th block may depend on elements of some previous blocks. Even if one assumes a CREW architecture, in which all the processors share some common memory space which can be accessed in parallel, this would still be essentially equivalent to a sequential model. This is so because elements dealt with by processor i at the beginning of block i may rely upon elements at the end of block $i-1$ which have not been processed yet by processor $i-1$; thus processor i can in fact start its work only after processor $i-1$ has terminated its own.

The easiest way to implement parallelization in spite of the above problem is to let each processor work independently of the others. The file is thus partitioned into n blocks which are encoded and decoded without any transfer of data between the processors. If the block size is large enough, this solution may even be recommendable: most LZ methods put a bound on the size of the history taken into account for the current item, and empirical tests show that the additional compression, obtained by increasing this history beyond some reasonable size, rapidly tends to zero. The cost of parallelization would therefore be a small deterioration in compression performance at the block boundaries, since each processor has to "learn" the main features of the file on its own, but this loss will often be tolerated as it may allow to cut the processing time by a factor of n. In [6] the authors suggest letting each processor keep the last characters of the previous block and thereby improve the encoding speed, but each block must then be larger than the size of the history window. On the other hand, putting a lower bound on the size N/n of each block effectively puts an upper bound on the number of processors n which can be used for a given file of size N, so we might not fully take advantage of all the available computing power.

We therefore turn to the question how to use n processors, even when the size of each block is not very large. In the next section we propose a new parallel coding algorithm, based on a time versus compression efficiency tradeoff which is related to the degree of parallelization. On the one extreme, for full parallelization, each of the n processors works independently, which may sharply reduce the compression gain if the size of the blocks is small. On the other extreme, all the processors may communicate, forcing delays that make this variant as time consuming as a sequential algorithm. The suggested tradeoff is based on a hierarchical structure of the connections between the processors, each of which depending at most on $\log n$ others. The task can be performed in parallel by n processors in $\log n$ sequential stages. There will be a deterioration in the compression ratio, but the loss will be inferior to that incurred when all n processors are independent.

In contrast to Huffman coding, for which parallel decoding could be applied regardless of whether the possibility of having multiple processors at decoding time was known at the time of encoding, there is a closer connection between encoding and decoding for LZ schemes. We therefore need to deal also with the parallel *encoding* scheme, and we assume that the same number of processors is available for both tasks.

Note, however, that one cannot assume simultaneously equi-sized blocks for both encoding and decoding. If encoding is done with blocks of fixed size, the resulting compressed blocks are of variable lengths. So one either has to store a vector of indices to the starting point of each processor in the compressed file, which adds an unnecessary storage overhead, or one performs *a priori* the compression on blocks of varying size, such that the resulting compressed blocks are all of roughly the same size. To get blocks of exactly the same size and to achieve byte alignment, one then needs to pad each block with a small number of bits, but in this case the loss of compression due to this padding is generally negligible. Moreover, the second alternative is also the preferred choice for many specific applications. For instance, in an Information Retrieval system built on a large static database, compression is done only once, so the speedup of parallelization may not have any impact, whereas decompression of selected parts is required for each query to be processed, raising the importance of parallel decoding.

2 A tree-structured hierarchy of processors

The suggested form of the hierarchy is that of a full binary tree, similarly to a binary heap. This basic form has already been mentioned in [6], but the way to use it as presented here is new. The input file is partitioned into n blocks $B_1, \ldots, B_n$, each of which is assigned to one of the available processors. Denote the n processors by $P_1, \ldots, P_n$, and assume, for the ease of description, that $n + 1$ is a power of 2, that is $n = 2^r - 1$ for some r. Processor P_1 is at the root of the tree and deals with the first block. As there is no need to "point into the future", communication lines between the processors may be unidirectional, permitting a processor with higher index to access processors with lower index, and in particular their local memories, but not vice versa. Restricting this to a tree layout yields a structure in which P_{2i} and P_{2i+1} can access the memory of P_i, for $1 \leq i \leq (n - 1)/2$. Figure 1 shows this layout for $n = 15$, the arrows indicating the dependencies between the processors. The numbers indicate both the indices of the blocks and of the corresponding processors.

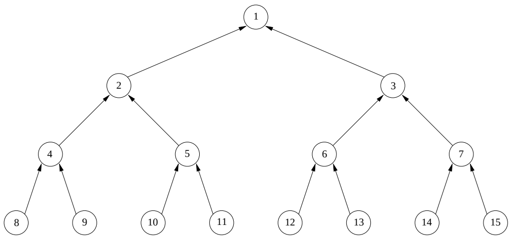


FIGURE 1: *Simple tree layout*

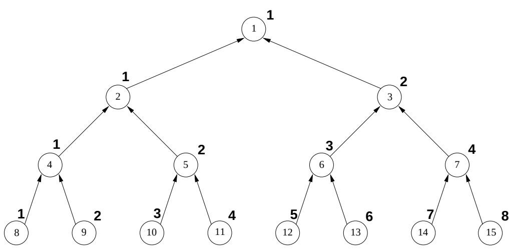


FIGURE 2: *Layer-by-layer layout*

The compression procedure for LZSS works as follows: P_1 starts at the beginning of block B_1, which is stored in its memory. Once this is done, P_2 and P_3 start simultaneously their work on B_2 and B_3 respectively, both searching for reoccurring strings first within the block they have been assigned to, and then extending the search back into block B_1. As mentioned above, P_2 can access the local memory of P_1 where B_1 is stored, without disturbing P_1's work. In general, after P_i has finished the processing of block B_i, processors P_{2i} and P_{2i+1} start scanning simultaneously their corresponding blocks. The compression of the file is thus not necessarily done layer by layer, e.g., P_{12} and P_{13} may start compressing blocks B_{12} and B_{13}, even if P_5 is not yet done with B_5.

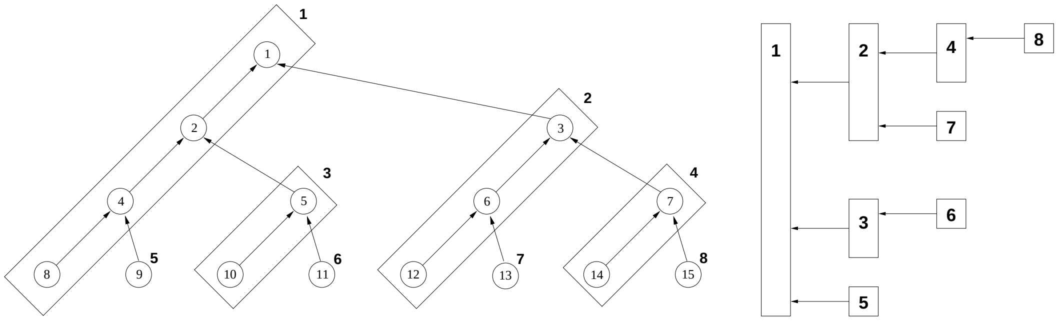
Note that while the blocks B_2 and B_1 are contiguous, this is not the case for B_3 and B_1, so that the (*off, len*) pairs do not necessarily point to *close* previous occurrences of a given string. This might affect compression efficiency, as one of the reasons for the good performance of LZ methods is the tendency of many files to repeat certain strings within the close vicinity of their initial occurrences. For processors and blocks with higher indices, the problem is even aggravated. The experimental section below brings empirical estimates of the resulting loss.

The layout suggested in Figure 1 is obviously wasteful, as processors of the higher layers stay idle after having compressed their assigned block. The number of necessary processors can be reduced by half, or, which is equivalent, the block size for a given number of processors may be doubled, if one allows a processor to deal with multiple blocks. The easiest way to achieve this is displayed in Figure 2, where the numbers in the nodes are the indices of the blocks, and the boldface numbers near the nodes refer to the processors. Processors $1, \ldots, 2^j$ are assigned sequentially, from left to right, to the blocks of layer j, $j = 0, 1, \ldots, r-1$. This simple way of enumerating the blocks has, however, two major drawbacks: refer, e.g., to block B_9 which should be compressed by processor P_2. First, it might be that P_1 finishes the compression of blocks B_2 and B_4, before P_2 is done with B_3. This causes an unnecessary delay, B_9 having to wait until P_2 processes both B_3 and B_5, which could be avoided if another processor would have been assigned to B_9, for example one of those that has not been used in the upper layers. Moreover, the problem is not only one of wasted time: P_2 stores in its memory information about the blocks it has processed, namely B_3 and B_5. But the compression of B_9 does not depend on these blocks, but only on B_4, B_2 and B_1. The problem thus is that the hierarchical structure of the tree is not inherited by the dependencies between the processors.

To correct this deficiency of the assignment scheme, each processor will continue working on one of the offsprings of its current block. For example, one could consistently assign a processor to the left child block of the current block, whereas the right child block is assigned to the next available newly used processor. More formally, let S_j^i be the index of the processor assigned to block j of layer i, where $i = 0, \ldots, r-1$ and $j = 1, \ldots, 2^i$, then $S_1^0 = 1$ and for $i > 0$ and $j = 1, \ldots, 2^{i-1}$,

$$S_{2j-1}^i = S_j^{i-1} \qquad \text{and} \qquad S_{2j}^i = 2^{i-1} + j.$$

The first layers are thus processed, from left to right, by processors with indices: (1), (1,2), (1, 3, 2, 4), (1, 5, 3, 6, 2, 7, 4, 8), etc. Figure 3(a) depicts the new layout of the blocks, the rectangles indicating the sets of blocks processed by the same processor. This structure induces a corresponding tree of processors, depicted in Figure 3(b).



(a) Tree of blocks

(b) Tree of processors

FIGURE 3: *New hierarchical structure*

As a results of this method, processor P_i will start its work with block B_{2i-1}, and then continue with B_{4i-2}, B_{8i-4}, etc. In each layer, the evenly indexed blocks inherit their processors from their parent block, and each of the oddly indexed blocks starts a new sequence of blocks with processors that have not been used before.

The memory requirements of the processors have also increased by this new scheme, and space for the data of up to $\log_2 n$ blocks has to be stored. However, most of the processors deal only with a few blocks. To evaluate the average number of blocks to be memorized, amortized over the m processors, suppose a full binary tree with r levels is used, so that there are $n = 2^r - 1$ nodes and $m = 2^{r-1} = (n+1)/2$ processors are needed. Then processor P_1 has to store information about r blocks, processor P_2 about $r-1$ blocks, the next two processors need only space corresponding to $r-2$ blocks, etc. The average amortized number of blocks to be referred to by a processor is therefore

$$\frac{1}{m}\left(r + \sum_{j=1}^{r-1}(r-j)2^{j-1}\right) = \frac{2^r - 1}{2^{r-1}} = 2 - \frac{1}{m},$$

that is, less than 2.

For the encoding and decoding procedures, we need a fast way to convert the index of a block into the index of the corresponding processor, i.e., a function

f, such that $f(i) = j$ if block B_i is coded by processor P_j. Define $r(i)$ as the largest power of 2 that divides the integer i, that is, $r(i)$ is the length of the longest suffix consisting only of zeros of the binary representation of i.

Claim:
$$f(i) = \frac{1}{2}\left(\frac{i}{2^{r(i)}} + 1\right).$$

Proof: By induction on i. For $i = 1$, we get $f(1) = 1$, which is correct. Assume the claim is true up to $i-1$. If i is odd, $r(i) = 0$ and the formula gives $f(i) = (i+1)/2$. As has been mentioned above, any oddly indexed block is the starting point of a new processor and indeed processor $P_{(i+1)/2}$ starts at block B_i. If i is even, block B_i is coded by the same processor as its parent block $B_{i/2}$, for which the inductive assumption applies, and we get

$$f(i) \;=\; f(i/2) \;=\; \frac{1}{2}\left(\frac{i/2}{2^{r(i/2)}} + 1\right) \;=\; \frac{1}{2}\left(\frac{i}{2\;2^{r(i)-1}} + 1\right) \;=\; \frac{1}{2}\left(\frac{i}{2^{r(i)}} + 1\right),$$

so that the formula holds also for i. ∎

2.1 Parallel coding for LZSS

We now turn to the implementation details of the encoding and decoding procedures for LZSS. Since the coding is done by stages, the parallel co-routines will invoke themselves the depending offsprings. For the encoding, the procedure *PLZSS-encode*(i, j) given in Figure 4 will process block B_i with processor P_j, where $j = f(i)$. The whole process is initialized by a call to *PLZSS-encode*(1,1) from the main program.

Each routine starts by copying the text of the current block into the memory of the processor, possibly adding to texts of previous blocks that have been stored there. As in the original LZSS, the longest substring in the history is sought that matches the suffix of the block starting at the current position. The search for this substring can be accelerated by several techniques, and one of the fastest is by use of a hash table, [13]. The longest substring is then replaced by a pair (*offset, length*), where *offset* is the distance (in characters) from the current position to the longest previous match, and *length* is the length of the match; if, however, *length* is too small (2 or 3 in implementations of [13], such as the patent [12], which is the basis of Microsoft's DoubleSpace), then a single character is sent to output and the current position is shifted by one to the right.

In our case, the search is not limited to the current block, but extends backwards to the parent blocks in the hierarchy, possibly up to the root. For example, referring to Figure 3, the encoding of block B_{13} will search also through

```
PLZSS-encode(i, j)
{
        append text of B_i to memory of P_j
        cur ⟵ 1
        while cur < |B_i|
        {
                S ⟵ suffix of B_i starting at cur
                ind ⟵ i
                while ind > 0
                {
                        access memory of P_f(ind) and
                        record occurrences in B_ind matching a prefix of S
                        ind ⟵ ⌊ind/2⌋
                }
                if longest occurrence not long enough
                {       encode single character        cur ⟵ cur + 1   }
                else
                {       encode as (offset, length)        cur ⟵ cur + len   }
        }
        perform in    ⎧ if 2i ≤ n         PLZSS-encode(2i, j)
        parallel      ⎩ if 2i + 1 ≤ n     PLZSS-encode(2i + 1, i + 1)
}
```

FIGURE 4: *Parallel LZSS encoding for block B_i by processor P_j*

B_6, B_3 and B_1, and thus access the memory of the processors P_7, P_2, P_2 and P_1, respectively. That is, the "text" in which earlier occurrences of substrings of B_{13} are searched is defined as the concatenation of the texts of blocks B_1, B_3, B_6 and B_{13}, though physically these texts are not contiguously stored. The values of *offset* refer to the distances in this concatenated text.

Note that the size of the history window is limited by some constant W in many implementations of LZSS. In our general description, we do not impose any such limit, but in fact, the encoding of any element is based on a history of size at most $\log_2 n\ \times$ the block size, where n is the number of blocks in the tree. Therefore, when the entire history is scanned to find the longest occurrence of a prefix of S, the scanning direction could be just as well top down rather than bottom up as in Figure 4. The reason for using a bottom up scan is that this applies also in the case the history window is limited; indeed, if only a part of the history is to be processed, it should be those blocks that are closest to S, to keep the values of *offset* as small as possible and because the main assumption of LZSS is that there is locality of reference.

For the decoding, recall that we assume that the encoded blocks are of equal size *Blocksize*. The decoding routine can thus address earlier locations as if the blocks, that are ancestors of the current block in the tree layout, were stored contiguously. Any element of the form (*offset, length*) in block B_i can point back into a block B_j, with $j = \lfloor i/2^b \rfloor$ for $b = 0, 1, \ldots, \lfloor \log_2 i \rfloor$, and the index of this block can be calculated by

$$b \longleftarrow \lceil (\mathit{offset} - \mathit{cur} + 1) / \mathit{Blocksize} \rceil,$$

where cur is the index of the current position in block B_i. The formal decoding procedure is given in Figure 5.

```
PLZSS-decode(i, j)
{
    cur ⟵ 1
    while there are more items to decode
    {
        if next item is a character
        {   store the character      cur ⟵ cur + 1   }
        else    // the item is (off, len)
        {
            if off < cur      // pointer within block B_i
                copy len characters, starting at position cur−off
            else     // pointer to earlier block
            {
                b ⟵ ⌈(off − cur + 1)/Blocksize⌉
                t ⟵ (off − cur) mod Blocksize
                copy len characters, starting at position t
                        in block B_⌊i/2^b⌋ which is stored in P_f(⌊i/2^b⌋)
            }
            cur ⟵ cur + len
        }
    }
    perform in    { if 2i ≤ n       PLZSS-decode(2i, j)
     parallel     { if 2i + 1 ≤ n   PLZSS-decode(2i + 1, i + 1)
}
```

FIGURE 5: *Parallel LZSS decoding for block B_i on processor P_j*

The input of the decoding routine is supposed to be a file consisting of a sequence of items, each being either a single character or a pointer of the form (*offset, length*); cur is the current index in the currently reconstructed block.

2.2 Parallel coding for LZW

Encoding and decoding for LZW is similar to that of LZSS, with a few differences. While for LZSS, the "dictionary" of previously encountered strings is in fact the text itself, LZW builds a continuously growing table $Table$, which need not be transmitted, as it is synchronously reconstructed by the decoder. The table is initialized to include the set of single characters composing the text, which is often assumed to be ASCII. If, as above, we denote by S the suffix of the text in block B_i starting at the current position, then the next encoded element will be the index of the longest prefix R of S for which $R \in Table$, and the next element to be adjoined to *Table* will be the shortest prefix R' of S for which $R' \notin Table$; R is a prefix of R' and R' extends R by one additional character.

During the encoding process of B_i, one therefore needs to access the tables in B_i itself and in the blocks which are ancestors of B_i in the tree layout, but the order of access has to be top down rather than in the LZSS case, for which the order can be either top down or bottom up, as explained earlier. For each i, we therefore need a list $list_i$ of the indices of the blocks accessed on the way from the root to block B_i, that is, $list_i[ind]$ is the number whose binary representation is given by the ind leftmost bits of the binary representation of i. For example, $list_{13} = [1, 3, 6, 13]$.

To encode a new element P, it is first searched for in *Table* of B_1, and if not found there, then in *Table* of $B_{list_i[2]}$, which is stored in the memory of processor $P_{f(list_i[2])}$, etc. However, storing only the elements in the tables may lead to errors. To illustrate this, consider the following example, referring again to Figure 3.

Suppose that the longest prefix of the string `abcde` appearing in the *Table* of B_1 is `abc`. Suppose we later encounter `abcd` in the text of block B_2. The string `abcd` will thus be adjoined to the same *Table*, since both B_1 and B_2 are processed by the same processor P_1. Assume now that the texts of both blocks B_5 and B_3 start with `abcde`. While for B_5 it is correct to store `abcde` as the first element in its *Table*, the first element to be stored in the *Table* of B_3 should be `abcd`, since the `abcd` in the memory of P_1 was generated by block B_2, whereas B_3 only depends on B_1.

To avoid such errors, we need a kind of a "time stamp", indicating at what stage an element has been added to a *Table*. If the elements are stored sequentially in these tables, one only needs to record the indices of the last element for each block. But implementations of LZW generally use hashing to maintain the tables, so one cannot rely on deducing information from its physical location, and each element has to be marked individually. The easiest way is to store with each string P also the index i of the block which caused the addition of P. This would require $\log_2 n$ bits for each entry. One can however take advantage of the fact that the elements stored by different blocks B_i in the memory of a given processor correspond to different indices ind in the corresponding lists $list_i$. It thus suffices to store with each element the index in $list_i$ rather than i itself, so that only $\log_2 \log_2 n$ bits are needed for each entry. The formal encoding and decoding procedures are given in Figures 6 and 7, respectively.

The parallel LZW encoding refers to the characters in the input block as belonging to a vector $B_i[cur]$, with cur giving the current index. If x and y

```
PLZW-encode(i, j)
{
    ω ⟵ B_i[1]
    cur ⟵ 2
    while cur ≤ |B_i|
    {
        ind ⟵ 1
        while list_i[ind] ≤ i
        {
            while  cur < |B_i| and
                   (ωB_i[cur], ind) ∈ Table stored in P_f(list_i[ind])
            {
                   ω ⟵ ωB_i[cur]
                   cur ⟵ cur + 1
                   last ⟵ ind
            }
            ind ⟵ ind + 1
        }
        indx ⟵ index(ω) in Table of P_f(list_i[last])
        store (indx, last) in memory of P_j
        store (ωB_i[cur], ind) in Table in memory of P_j
        ω ⟵ B_i[cur]
        cur ⟵ cur + 1
    }
    perform in   { if 2i ≤ n       PLZW-encode(2i, j)
     parallel    { if 2i + 1 ≤ n   PLZW-encode(2i + 1, i + 1)
}
```

FIGURE 6: *Parallel LZW encoding for block B_i on processor P_j*

are strings, then xy denotes their concatenation. As explained above, since the *Table* corresponding to block B_i is stored in the memory of a processor which is also accessed by other blocks, each element stored in the *Table* needs an identifier indicating the block from which it has been generated. The elements in the *Table* are therefore of the form (*string, identifier*).

The output of LZW encoding is a sequence of pointers, which are the indices of the encoded elements in the *Table*. In our case, these pointers are of the form (*index, identifier*). There is, however, no deterioration in the compression efficiency, as the additional bits needed for the identifier are saved in the representation of the index, which addresses a smaller range.

For simplicity, we do not go into details of handling the incremental encoding of the indices, and overflow conditions when the *Table* gets full. It can be done as for the serial LZW.

The parallel LZW decode routine assumes that its input is a sequence of elements of the form (*index, identifier*). The empty string is denoted by Λ. The algorithm in Figure 7 is a simplified version of the decoding, which does

PLZW-decode(i, j)

```
{
    cur ⟵ 1
    old ⟵ Λ
    while cur ≤ number of items in block B_i
    {
        (indx, ind) ⟵ B_i[cur]
        access Table in P_{f(list_i[ind])} at index indx
            and send string str found there to output
        if old ≠ Λ
            store (old first[str], ⌈log_2(i+1)⌉) in Table of P_j
        old ⟵ str
        cur ⟵ cur + 1
    }

    perform in parallel { if 2i ≤ n       PLZW-decode(2i, j)
                        { if 2i + 1 ≤ n   PLZW-decode(2i + 1, i + 1)
}
```

FIGURE 7: *Parallel LZW decoding for block B_i on processor P_j*

not work in case the current element to be decoded was the last one to be added to the *Table*. This is also a problem in the original LZW decoding and can be solved here in the same way. The details have been omitted to keep the emphasis on the parallelization.

3 Higher order trees

In this section, we wish to explore possible tradeoffs that can be achieved by generalizing the binary tree layout to trees of higher order $k > 2$, in which each node has k children. Once a processor is done with a given block, it will start to work on the block's leftmost child, while $k - 1$ new processors will start their work on the remaining offsprings. Passing to higher order trees may yield several advantages. For instance, the depth of a k-ary tree is only $\log_k n$, so that the chain of dependencies is shorter than in the binary case, and thus less information need be stored per processor. Moreover, after the ith parallel step, the number of blocks that have been dealt with is $\sum_{j=1}^{i} k^{j-1}$, so a given block is reached faster when k is larger.

To measure the level of exploitation of the m available processors, define a utilzation factor as the average fraction of the processors which are active. At the lowest level of the tree, all the processors are busy; at the level just above the lowest, only $\frac{1}{k}$ of the processors are active, etc. It would thus seem, at first sight, that if we assume that each level has the same expected execution time, the average utilization factor would be proportional to $\sum \left(\frac{1}{k}\right)^i \longrightarrow 1 + \frac{1}{k-1}$, which is a decreasing function of k. But this did not take into account that the number of levels decreases when k increases. The average time spent on each level being $\frac{1}{\log_k n}$, we get that the average utilization factor is

$$\frac{1}{\log_k n} \sum_{i=0}^{\log_k n} \left(\frac{1}{k}\right)^i \quad \longrightarrow \quad \frac{1}{\log_2 n} \; \frac{k \log_2 k}{k-1},$$

which is an increasing function of k for $k \geq 2$, suggesting that a higher order tree layout may be advantageous for better utilization of the available resources.

The average number of blocks to be memorized, amortized over the m processors, is evaluated as follows. One processor works on level 0, $k-1$ additional ones on level 1, $k(k-1)$ more are added at the next level, etc. The total number of processors is therefore

$$m = 1 + (k-1) \sum_{j=0}^{r-2} k^j \quad = \quad k^{r-1}.$$

Processor P_1 has to store information about r blocks, processor P_2 to P_k about $r-1$ blocks, the next $k(k-1)$ processors need only space corresponding to $r-2$ blocks, and the next $k^2(k-1)$ processors only to $r-3$ blocks, etc. The total required space, when summed over all the processors, is then

$$r + \sum_{j=1}^{r-1} (r-j) k^{j-1} (k-1) \quad = \quad \frac{k^r - 1}{k-1}.$$

Amortizing this space over the m processors, we get as average required memory per processor:

$$\frac{k^r - 1}{(k-1)k^{r-1}} \quad = \quad 1 + \frac{1 - \frac{1}{m}}{k-1},$$

which is decreasing with k. So from the point of view of local space requirements, it is also worth passing to higher order trees.

However, all these advantages calling for larger k are counterbalanced by the fact that with increasing k, the hierarchical layout tends increasingly to be equivalent to using m independent processors, affecting the compression efficiency when the block size is small. Indeed, the LZ compression schemes take advantage of the fact that certain strings tend to reoccur shortly after a first appearance, and this locality of reference is disturbed by connecting blocks which are not adjacent. In our case, for a fixed block size, the distance, in the file, between blocks treated by the same processor, is increasing with k, so we might expect better compression with lower k. In the next section, we bring empirical results comparing the compression performance for various values of k.

In a straightforward generalization of the binary case, the blocks would be numbered sequentially top down, left to right, so that the children of block B_i would be the blocks $B_{k(i-1)+1+t}$ for $t = 1, \ldots, k$. The correspondence between blocks and processors would then be given as follows: if S^i_j is the index of the processor assigned to block j of layer i, where $i = 0, \ldots, r-1$ and $j = 1, \ldots, k^i$, then $S^0_1 = 1$ and

$$\text{for } i > 0 \text{ and } j = 1, \ldots, k^{i-1}, \qquad S^i_{k(j-1)+1} = S^{i-1}_j$$

$$\text{and} \quad \text{for } t = 2, \ldots, k \qquad S^i_{k(j-1)+t} = k^{i-1} + (k-1)(j-1) + t - 1.$$

For example, for $k = 3$, we would get as order of processors, from left to right, for the first layers:: (1), (1, 2, 3), (1, 4, 5, 2, 6, 7, 3, 8, 9), etc. Figure 8 depicts this layout of the blocks, for $k = 3$, on a tree with 4 layers, in similar form as in Figure 3(a).

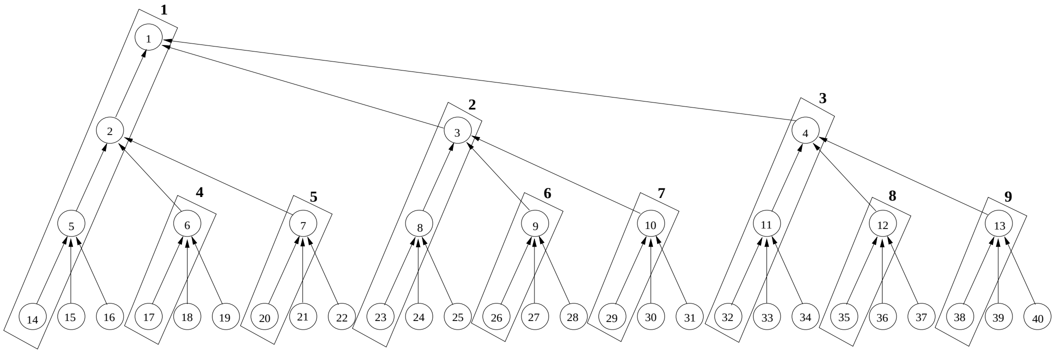


FIGURE 8: *Hierarchical structure of ternary tree*

As above for the binary case, we would need a function $f_k(i)$ converting the index of a block into that of the corresponding processor for the k-ary tree, i.e., $f_k(i) = j$ if block B_i is coded by processor P_j. This function would be given by

$$f_k(i) = \begin{cases} f_k\left(\dfrac{i+k-2}{k}\right) & \text{if} \quad i \bmod k = 2 \\ i - \left\lceil \dfrac{i-2}{k} \right\rceil & \text{if} \quad i \bmod k \neq 2 \end{cases}$$

The particular case $k = 2$ coincides with the formula given earlier if one interprets the condition $i \bmod 2 = 2$ as standing for i is even. Indeed, one gets then that $f_2(i) = f_2(i/2)$ for even i, and $f_2(i) = (i+1)/2$ if i is odd.

However, with a sequential numbering of the blocks, the parent-child relations of the blocks are not trivially obvious from their indices. This is a disadvantage, since one needs a direct way to address ancestors in the LZ coding routines. One could of course prepare for each index i of a block, a list $list_i$ as suggested above in Section 2.2, giving the sequence of the indices of the blocks accessed from the root to block B_i; but for the ternary case, we would get, for example, $list_{21} = [1, 2, 7, 21]$, which is not trivially related to the ternary representation of the index 21, as $list_i$ for the binary case was related to the binary representation of i.

To rectify this deficiency, the following new numbering of the blocks in a k-ary layout is suggested: the blocks in layer i, $i = 0, \ldots, r-1$, will be indexed from left to right by $k^i + j$, $j = 0, \ldots, k^i - 1$. For example, for $k = 3$, the sequence of indices will be $1, 3, 4, 5, 9, 10, \ldots, 17, 27, 28, \ldots, 53, 81, 82, \ldots$. The main property of this way of enumerating the blocks is that the following relation holds between a block and its offsprings: if α is the representation of the index of the given block in the standard k-ary numeration system, then the representations of the indices of the k children of this block are $\alpha 0, \alpha 1, \ldots, \alpha(k-1)$. Note that for $k = 2$, the new numbering coincides with the sequential numbering of Section 2.

Another way to look at it is by considering the layout as a full k-ary trie, labelled in a similar way as suffix trees: the edges emanating from a given node are labelled, from left to right, by $0, 1, \ldots, k-1$, the root is labelled by the empty string Λ, and each node x is labelled by the concatenation of the labels on the edges of the unique path from the root to x. Here we have merely prefixed each of these node-labels by a leading 1, to avoid ambiguities when the labels are considered as numbers rather than k-ary strings. Without the leading 1s, the labels of the nodes of the leftmost branch of the tree would be $\Lambda, 0, 00, 000, \ldots$, prefixing the 1 turns them into different numbers $1, 10, 100, \ldots$. As an example, consider the block B_{897} in a 5-ary tree. The chain of blocks leading to it is $B_1, B_7, B_{35}, B_{179}$ and B_{897}, and their indices, in 5-ary, are 1, 12, 120, 1204, 12042, respectively.

One can therefore readily generalize the binary based LZ coding routines by noting that the ancestors of block B_i are the blocks $B_{\lfloor i/k^b \rfloor}$, for $b = 1, 2, \ldots$. The new definition of the function $f_k(i)$, giving the index of the processor dealing with block B_i is as follows: let $t(i) = \lfloor \log_k i \rfloor$ be the length of the k-ary representation of i not including the leading 1, so that $t(i)$ is in fact the index of the layer in which block B_i occurs, and let $r(i)$, as above, be the length of the longest suffix consisting only of zeros in the k-ary representation of i.

Claim:
$$f_k(i) = \lfloor k^{t(i)-r(i)-1} \rfloor + (k-1) \left\lfloor \frac{i - k^{t(i)}}{k^{r(i)+1}} \right\rfloor + \frac{i}{k^{r(i)}} \bmod k.$$

Proof: By induction on the relevant values of i. For $i = 1$, the first component is $\lfloor k^{0-0-1} \rfloor = 0$ (in fact, the *floor* operator is only needed in this special case, as for $i > 1$, this component will always be an integer), the second component is 0 and the third is 1, so we get $f_k(1) = 1$ for all k.

Assume the claim is true up to $i - 1$ and consider first a node with index $i > 1$

to which a new processor is assigned; the index i of this node is then such that $i \bmod k \neq 0$, so that $r(i) = 0$. The node appears on level $t(i)$ in the tree and the number of processors used in the $t(i)$ levels above the current one is $k^{t(i)-1}$, which accounts for the first component. The relative index of node i within layer $t(i)$ is $i - k^{t(i)}$. This layer can be partitioned into groups of k nodes, each group including the child nodes of one of the nodes of layer $t(i) - 1$. Since we assume here that $i - k^{t(i)}$ is not divisible by k, the number of groups to the left of the one to which node i belongs is $\lfloor (i - k^{t(i)})/k \rfloor$, and each such group contributes $k - 1$ new processors, as only the first node in each group inherits the processor of the parent node; this accounts the for the second component. What still need to be added is the relative index of node i within the group it belongs to, and this index is $i \bmod k$.

If i is a multiple of k, then B_i is dealt with by the same processor as its parent node $B_{i/k}$. Noting that $t(i/k) = t(i) - 1$, $r(i/k) = r(i) - 1$, and that we can apply the inductive assumption for $i/k < i$, we get that

$$f_k(i) \;=\; f_k(i/k) \;=\; k^{t(i)-1-(r(i)-1)-1} + (k-1)\left\lfloor \frac{i/k - k^{t(i)-1}}{k^{(r(i)-1)+1}} \right\rfloor + \frac{i/k}{k^{(r(i)-1)}} \bmod k,$$

and the right hand side reduces to the formula given in the claim, which shows that it holds also for i. ∎

A way relating the function $f_k(i)$ to the k-ary representation of i is the following: first, delete the longest suffix consisting only of zeros; define A as the remaining string from which the rightmost k-ary digit, denoted C, has been removed, and define B as the string obtained from A by removing its leading 1. Then

$$f_k(i) = A + (k-2)B + C.$$

Returning to the above example, we get $f_5(897) = 1204_5 + 3 \cdot 204_5 + 2 = 343$.

The generalizations of the LZ coding routines given in the previous section, both for LZSS and LZW, both encode and decode, are now straightforward. In particular, there are k parallel recursive calls of the form

$$\text{perform in parallel} \begin{cases} \text{if } ki \le n & \mathit{PLZ\text{-}code}(ki, f_k(ki)) \\ \text{if } ki+1 \le n & \mathit{PLZ\text{-}code}(ki+1, f_k(ki+1)) \\ \quad\vdots & \\ \text{if } ki+k-1 \le n & \mathit{PLZ\text{-}code}(ki+k-1, f_k(ki+k-1)) \end{cases}$$

4 Experimental results

We now report on some experiments on files in different languages: the Bible (King James Version) in English, the Bible in Hebrew and the *Dictionnaire philosophique* of Voltaire in French. Table 1 first brings the sizes of the files in MB and to what size they can be reduced by LZSS and LZW, expressed in percent of the sizes of the original files. We consider three algorithms: the serial one, using a single processor and yielding the compressed sizes in Table 1, but being slow; a parallel algorithm we refer to as *standard*, where each block is treated independently of the others; and the *new* parallel algorithm presented herein, with $k = 2$, which exploits the hierarchical layout. The columns headed Time in Table 1 compare the new algorithm with the serial and the standard parallel ones. The time measurements were taken on a Sun 450 with four UltraSPARC–II 248 MHz processors sharing a common memory, which allowed a layout with 7 blocks. For the serial algorithms, the code provided by [9] has been used, with a maximal dictionary size of 32K for LZW and a history buffer of 4K for LZSS. The values are in seconds and correspond to LZW, which turned out to give better compression performance than LZSS in our case. The improvement is obviously not expected to be 4-fold, due to the overhead of the parallelization, but on the examples the time is generally cut to less than half.

	Size			Time					
	Full	compressed by		compression			decompression		
		LZSS	LZW	Serial	Stand.	New	Serial	Stand.	New
Eng Bib	3.860	41.6	36.6	5.508	1.513	2.296	3.653	1.081	1.504
Heb Bib	1.471	51.7	44.7	2.134	0.645	0.853	1.488	0.382	0.566
Voltaire	0.529	49.0	40.6	0.770	0.227	0.380	0.456	0.190	0.310

TABLE 1: *Size and time measurements on test files*

For the compression performance, we first compare the standard parallel version with the new one for $k = 2$. Both are equivalent to the serial algorithm if the block size is chosen large enough, as in [6]. The graphs in Figure 9 show the sizes of the compressed files in MB as functions of the block size (in bytes), for both LZSS and LZW. We see that for large enough blocks (larger than the history buffer) the loss relative to a serial algorithm with a single processor is negligible (about 1%) for both the standard and the new methods. However, when the blocks become shorter, the compression gain in the independent model almost vanishes, whereas with the new processor layout the decrease in compression performance is much slower. For blocks as small as 128 bytes, running a standard parallel compression achieves only about 1–4% compression for LZSS and about 12–15% for LZW, while with the new layout this might be reduced by some additional 30–40%.

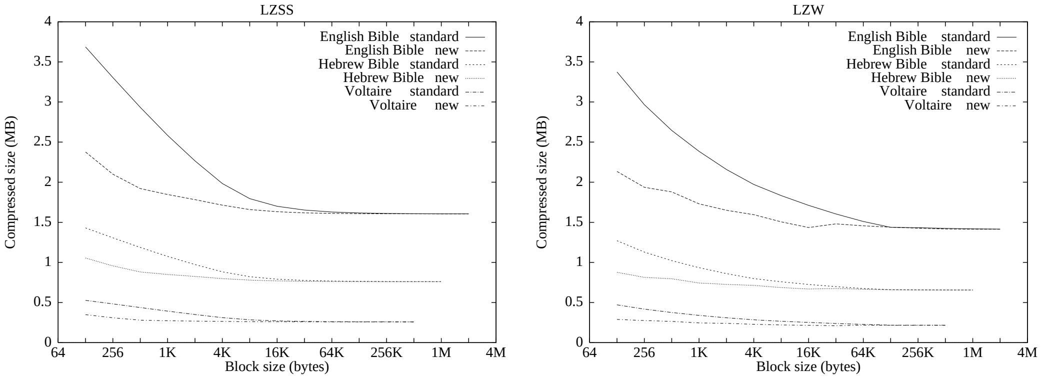


FIGURE 9: *Size of compressed file as function of block size*

The graphs in Figure 10 compare the compression performance of the higher order layouts corresponding to $3 \leq k \leq 5$, with those of the binary layout and with the standard parallel algorithm, using the English Bible file as example. As expected, for LZSS, the compression gets worse with increasing k, for all block sizes, and for fixed k, compression is a decreasing function of the block size, for all k. For LZW this is also the general trend, though there are small fluctuation. Interestingly, for the smaller block sizes, the graphs of the hierarchic methods, even with $k = 5$, are much closer to each other than they are to the graph of the standard parallel method, which implies that higher order layouts might be worth looking at if small blocks are required. A possible reason for the difference between LZSS and LZW is that in the former, blocks are processed bottom up in our implementation, so that for fixed block size but with increasing k, the referenced reoccurring strings are farther away, thus tend to yield lower savings. The same would be true also for LZW, but for it, processing has been done top down; if the first few blocks are representative of the whole file, they will contain "good" strings to be used in subsequent blocks, so compression might be less affected by the choice of k than in the LZSS case.

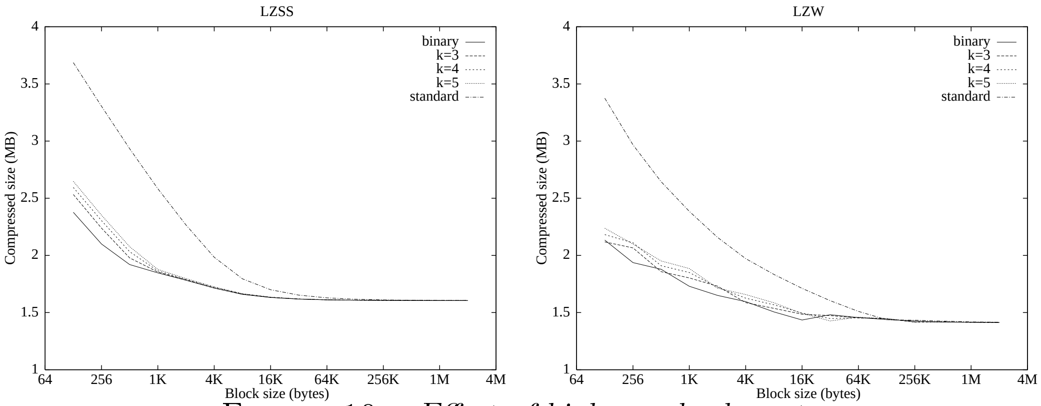


FIGURE 10: *Effect of higher order layouts*

We conclude that the simple hierarchical layout might allow us to considerably reduce the size of the blocks that are processed in parallel without paying too high a price in compression performance. As a consequence, if a large number of processors is available, it enables a better utilization of their full combined computing power.

References

[1] Belinskaya D., De Agostino S., Storer J.A., Near Optimal Compression with Respect to a Static Dictionary on a Practical Massively Parallel Architecture, *Proc. Data Compression Conference DCC–95*, Snowbird, Utah *IEEE Computer Society Press* (1995) 172–181.

[2] De Agostino S., Storer J.A., Parallel Algorithms for Optimal Compression using Dictionaries with the Prefix Property, *Proc. Data Compression Conference DCC–92*, Snowbird, Utah *IEEE Computer Society Press* (1992) 52–61.

[3] Gonzalez Smith M.E., Storer J.A., Parallel Algorithms for Data Compression, *Journal of the ACM* **32**(2) (1985) 344–373.

[4] Hirschberg D.S., Stauffer L.M., Parsing Algorithms for Dictionary Compression on the PRAM, *Proc. Data Compression Conference DCC–94*, Snowbird, Utah *IEEE Computer Society Press* (1994) 136–145.

[5] Howard P.G., Vitter J.S., Parallel lossless image compression using Huffman and arithmetic coding, *Proc. Data Compression Conference DCC–92*, Snowbird, Utah (1992) 299–308.

[6] Iwata K., Morii M., Uyematsu T., Okamoto E., A simple parallel algorithm for the Ziv-Lempel encoding, *IEICE Trans. Fundamentals* **E81–A** (1998) 709–712.

[7] Klein S.T., Wiseman Y., Parallel Huffman decoding with applications to JPEG files, *The Computer Journal* **46**(5)(2003) 487–497.

[8] Larmore L.L., Przytycka T.M., Constructing Huffman Trees in Parallel, *SIAM Journal of Computing* **24**(6) (1995) 1163–1169.

[9] Nelson M., *The Data Compression Book*, M & T Publishing, Inc., (1991)

[10] Storer J.A., Szymanski, T.G., Data compression via textual substitution, *Journal of the ACM* **29** (1982) 928–951.

[11] Welch T.A., A technique for high-performance data compression, *IEEE Computer* **17** (1984) 8–19.

[12] Whiting D.L., George G.A., Ivey G.E., Data Compression Apparatus and Method, U.S. Patent 5,126,739 (1992).

[13] Williams R.N., An extremely fast Ziv-Lempel data compression algorithm, *Proc. Data Compression Conference DCC–91*, Snowbird, Utah (1991) 362–371.

[14] Ziv J., Lempel A., A universal algorithm for sequential data compression, *IEEE Trans. on Inf. Th.* **IT–23** (1977) 337–343.

[15] Ziv J., Lempel A., Compression of individual sequences via variable-rate coding, *IEEE Trans. on Inf. Th.* **IT–24** (1978) 530–536.