Parallel Huffman Decoding

EXTENDED ABSTRACT

S. T. Klein
Dept. of Math. & CS
Bar Ilan University
Ramat-Gan 52900
Israel
tomi@cs.biu.ac.il

Y. Wiseman
Dept. of Math. & CS
Bar Ilan University and
Jerusalem College
of Technology
wiseman@cs.biu.ac.il

Abstract: A simple parallel algorithm for decoding a Huffman encoded file is presented, exploiting the tendency of Huffman codes to resynchronize quickly in most cases. An extention to JPEG decoding is mentioned.

1. Introduction

Huffman coding is still one of the popular compression techniques and is widely used by itself [18, 2] or in connection with other methods such as JPEG [17]. Huffman's original method [11] is not adaptive and needs two passes over the data to be compressed. This might be a disadvantage in certain applications, in which dynamic algorithms, such as those based on the works of Lempel and Ziv [19, 20], are the preferred choice. There are, however, situations, in which a static method is required:

- A large static Information Retrieval (IR) System [1] is compressed only once, but many short passages have to be decompressed on demand in response to a query; each such passage should therefore be decodable on its own, ruling out compression methods which are based on the knowledge of earlier blocks.
- As more texts will be stored on the Web in compressed form, methods for searching for patterns directly in the compressed text will gain importance [16]. The idea will then be to compress the pattern and then scan the compressed text, rather than decompressing the text and search for the original pattern within it. This form of *compressed matching* will be facilitated if every text item is always compressed in the same way, which is not the case for dynamic methods.
- When more than one processor is available, a static compression scheme may allow the decoding of several data pieces in parallel.

It is on the third point that we shall concentrate in this paper. We shall explore a method allowing the parallel decoding of a file that has been compressed by a static Huffman code, exploiting in particular the tendency of Huffman codes to resynchronize quickly in case of an error.

Previous work on parallelizing compression includes [3, 4, 9], which deal with LZ compression, and [10]. A parallel method for the construction of Huffman trees can be found in [15]. Our focus is on *decompression*, because it may be more important than compression in some cases. For instance, in IR applications as the one mentioned in the first point above, compression is done only once and may therefore be as time consuming as necessary, but decompression of short pieces is done on-line and ought to be fast to allow a reasonable response time to a query.

In the next section we review the main problem faced by parallel decompression, namely synchronization. Section 3 then presents the algorithm and Section 4 some experimental results. Finally, we show how to apply the method also to lossy JPEG compression.

2. Synchronization

When more than one processor is available at decompression time, the compressed text can be split into blocks, and each processor can be assigned one of the blocks for decompression. The problem is of course that the sizes of the blocks are fixed in advance, and since Huffman codewords have variable length, a block-boundary does not necessary coincide with a codeword boundary. But Huffman codes are complete, which means that any binary sequence can be "decoded" as if it were the encoding of some text, so that synchronization errors may go undetected. Consider for example the simple Huffman code $\{00, 010, 011, 10, 11\}$ for the characters `A,B,C,D,E`, respectively. The encoding of the string `BACEAD` would then be the binary string 01000011110010. Suppose that one of the processors would be assigned a block starting at the third bit of this string. It would then decode the block as `AAEEAD`, the first three characters of which are erroneous.

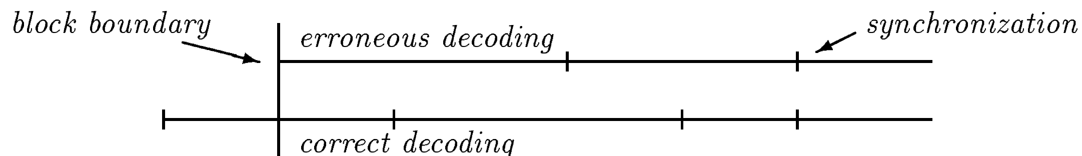

FIGURE 1: *Schematic representation of parallel decoding*

The general situation is depicted in Figure 1. The upper line symbolizes the decoding which starts at the block boundary and might therefore produce an erroneous decoding for several codewords. The line below shows the correct decoding: the block boundary could have occurred within a codeword, so that the following block starts

with some proper suffix of a codeword. Typically, the correct and the erroneous decodings could then generate different output sequences, up to a position in the binary string to be decoded, which, for both processes, holds a bit that completes a codeword of the given code. This position is indicated as *synchronization* point, as subsequent bits will be correctly decoded in any case.

Synchronization points do not always exist. A simple example would be a fixed length code, for which every codeword has length k bits. If the blocksize is not a multiple of k, all the codewords of the second block will be out of synchronization. But in the case of a fixed length code, the blocksize could be chosen *a priori* as a multiple of the codeword length. Moreover, fixed length codes are optimal, from the compression point of view, only for nearly uniform distributions.

On the other hand, there are also variable length codes for which synchronization will not be achieved. Refer again to Figure 1, and denote by x and y, respectively, the last codeword before the synchronization point for the erroneous and the correct decoding. Then either y has to be a suffix of x (as in the example in the figure), or x is a suffix of y. In either case, the code cannot have the so-called *suffix-property*, asserting that no codeword can be the suffix of any other, similarly to the well-known *prefix-property* of all Huffman codes. Accordingly, codes having both the prefix and the suffix property have been called *never-self-synchronizing* in [8]; they are called *affix* codes in [6]. There are infinitely many different complete variable-length affix codes, e.g., $\{01, 000, 100, 110, 111, 0010, 0011, 1010, 1011\}$, but they are nonetheless extremely rare [7]. For none of the real-life distributions we checked could an affix code be constructed. For those rare artificial distributions for which it was possible, the affix code had to be carefully designed; selecting the code in some systematic way or using canonical codes [13] did not yield affix codes.

For certain distributions, a Huffman code may be constructed that includes synchronizing codewords or sequences [5, 14]. These are codewords or sequences after the occurrence of which decoding will be correct, regardless of any possible error before them. The higher the probability of these codewords, the lower the expected number of falsely decoded bits at the beginning of each block, so the techniques of [5] may be applied to improve the performance of the parallel Huffman decoding. In practice, however, synchronization is fast even without the help of synchronizing codewords.

3. Parallel Decoding

The basic idea of the parallel decoding algorithm is letting the processor i, which has been assigned to decode block i, overflow and continue decoding in the consecutive block $i+1$, until a synchronization point is reached. Assuming that the last codewords in block i are already correctly decoded, processor i will give the correct decoding of the first few codewords in block $i+1$. Once a synchronization point in block $i+1$ is detected, processor i can stop (or be reassigned to the decoding of another block), since the remaining bits in block $i+1$ have been correctly decoded by processor $i+1$. In particular, the synchronization point can be immediately at the block boundary,

in case the last codeword of the previous block happens to fit there in its entirety.

If the assumption that the last codewords in block i have been correctly decoded by processor i is not true, the synchronization point found in block $i+1$ is worthless. However, in this case, processor $i-1$ has not been able to find a synchronization point in block i, and did therefore continue working also on block $i+1$. The correctness is now based on the assumption that the last codewords in block $i-1$ have been correctly decoded. This argument can be extended to $i-2$, etc., but ultimately, there must be a block j, with $j < i$, for which this is true, since processor 1 starts at the beginning of the file and its output is correct. Therefore, in the worst case, any output produced by all the processors i, with $i > 1$, is useless, and the parallel decoding reduces to a sequential one by processor 1 alone. As mentioned above, such a worst case behaviour seems to be extremely rare, as in most cases, the synchronization points are found quickly, long before the end of the block.

The formal parallel decoding algorithm for processor i is given in Figure 2. Processor i maintains a vector V_i, which is also accessible to processors j, for $j < i$, and records the indices, within block i, of the last bit of each codeword. In general, the first few elements of V_i will be wrong, corresponding to the erroneous decoding at the beginning of the block, but they will be corrected when processor $i-1$ moves into block i. This vector V_i also serves as indicator for processor i to stop: as soon as a value is detected that is equal to one of the values stored in V_i by an earlier processor, synchronization has been achieved. We use the abbreviations EOB for end of block and eoc for end of codeword.

To get an estimate of the number of bits that have to be processed before a synchronization point is found, we introduce the following notations. Let $\mathcal{T}$ denote the Huffman tree corresponding to a given Huffman code. The elements which are encoded appear with probabilities $p_1, \ldots, p_n$ in the text, and the lengths of the corresponding Huffman codewords are $\ell_1, \ldots, \ell_n$, respectively. We shall also use the notation p_y for the probability of the element corresponding to the leaf y. Denote by $\mathcal{L}$ the set of the leaves of $\mathcal{T}$, and by $\mathcal{I}$ the set of its internal nodes. For each $x \in \mathcal{I}$, we define $\mathcal{T}_x$ as the subtree of $\mathcal{T}$ rooted at x, and we denote by $\mathcal{L}_x = \mathcal{L} \cap \mathcal{T}_x$ the set of its leaves. The internal nodes $\mathcal{I}$ correspond to the positions at which a codeword might be cut by a block-boundary. In particular, the root r of the tree, which belongs to $\mathcal{I}$, corresponds to the special case where the block-boundary falls between two codewords.

We assume that a block boundary occurs at random in any possible position, that is, at any internal node of $\mathcal{T}$. This is an approximation, since in certain cases, not all the positions are possible cut-points, nor do those that are possible all appear with the same probability. For example, if both the block-size and all the codeword lengths are even, then no codeword can be cut by a block boundary after an odd number of bits. But for many real-life distributions, especially for the large ones with thousands or even millions of elements, the corresponding Huffman codes have codewords of all possible lengths in a certain range; adding to this the fact that the block size is generally chosen so as to accomodate a very large number of consecutive codewords, we conclude that our assumption can be justified.

```
Start decoding at beginning of block i
Record indices of end of codewords in vector V_i
Continue until EOB

If   EOB is an eoc   STOP
else                 // overflow to next block
{
        i <-- i+1
        k <-- 1
        repeat
        {
                decode to next eoc
                if   this eoc is EOB   STOP
                if   EOB was passed
                {       i <-- i+1       k <-- 1       }
                else
                {
                        j <-- index of eoc in block i
                        while   V_i[k] < j
                        {
                                erase V_i[k]
                                k <-- k+1
                        }
                        if   V_i[k] = j   STOP
                        else   insert j in front of V_i[k] in V_i
                }
        }
}
```

FIGURE 2: *Decoding algorithm for processor i*

Consider the fact of having a block boundary in a certain position as if it were generated by the following random process: the compressed text consisting of a given sequence of concatenated codewords, we "throw" at random boundaries into this string, that is, we pick randomly bit positions which shall act as the starting positions of the blocks. In this sense, we can speak about the probability of having a block boundary in a certain position. For a given internal node $x \in \mathcal{I}$, the probability $P(x)$ of the position corresponding to x being picked as a boundary point will be proportional to $p_i\ell_i$, and not just to p_i, since we deal with a random process on the compressed text and not on the original one. Each leaf of the Huffman tree is associated with a probability p_i, and the probability associated with an internal node y is the sum of the probabilities associated with the two children of y. Thus, when adding the probabilities associated with all the internal nodes, we get $W = \sum_{i=1}^{n} p_i\ell_i$, the weighted average codeword length, and the probability $P(x)$ is given by

$$P(x) = \frac{\sum_{y \in \mathcal{L}_x} p_y}{W}.$$

This is indeed a probability distribution, as $\sum_{x \in \mathcal{I}} P(x) = 1$. For $x \in \mathcal{I}$ and $y \in \mathcal{L}_x$, define

$$Q(x,y) = \begin{cases} 1 & \text{if the path from } x \text{ to } y \text{ corresponds to a sequence} \\ & \quad \text{of one or more codewords in the code} \\ 0 & \text{otherwise,} \end{cases}$$

that is, $Q(x,y) = 1$ if and only if, in case a codeword has been cut by a block boundary, synchronization is reestablished at the end of this codeword. In particular, for an affix code, $Q(x,y) = 0$ for all x and y, unless x is the root.

For a given block starting at some internal bit of a codeword c, let $\mathcal{S}$ denote the event that the synchronization point is already at the end of c, i.e., only the codeword cut by the boundary is lost, if at all, and the subsequent ones will be correctly recognized by the processor assigned to this block. We evaluate the probability $P(\mathcal{S})$ by conditioning on the position of the possible cut-points:

$$P(\mathcal{S}) = \sum_{x \in \mathcal{I}} P(\mathcal{S} \mid \text{cut-point is at } x)\ P(x).$$

But $P(\mathcal{S} \mid \text{cut-point is at } x)$ is the weighted average of the decoding successes, summed over all the leaves of $\mathcal{T}_x$, that is

$$P(\mathcal{S} \mid \text{cut-point is at } x) = \frac{\sum_{y \in \mathcal{L}_x} p_y\ Q(x,y)}{\sum_{y \in \mathcal{L}_x} p_y},$$

from which we get that

$$P(\mathcal{S}) = \frac{\sum_{x \in \mathcal{I}} \sum_{y \in \mathcal{L}_x} p_y\ Q(x,y)}{W}. \tag{1}$$

We therefore conclude that the probability $P(\mathcal{S})$, which we shall denote below P_1, depends only on the given distribution and on the shape of the Huffman tree. The more paths from internal nodes to the leaves match other such paths starting at the root, the more $Q(x,y)$s will be 1 and the higher $P(\mathcal{S})$ will be. A good choice for the shape seems then to be a *canonical* tree, in which the leaves appear, from left to right, in non-decreasing order of their depths [13]. Such a shape tends to favor reoccuring structure patterns. Returning to the example of the affix code above, the canonical Huffman code with the same codeword lengths is $\{00, 010, 011, 100, 101, 1100, 1101, 1110, 1111\}$. For this tree, we have $Q(x,y) = 1$ if x is the root or if x is the internal node corresponding to the prefix 1 and y is one of the leaves corresponding to 100, 1100 or 1101; or if x corresponds to 11 and y to 1100; for all other (x,y) pairs, $Q(x,y) = 0$.

Consider now the case when the complementary event of $\mathcal{S}$ occurs, that is, synchronization was not regained at the end of the first codeword. But we are then in a similar situation: a decoding process is started at some internal position within a codeword c and we ask what is the probability to resynchronize at the end of c. If the number of codewords in a block is large enough, we may assume that this event is

independent of the previous one, so we again get the same probability $P(\mathcal{S})$. Extending this argument, we see that the *number* of codewords c we have to process until success, i.e., synchronization, is geometrically distributed, and its expected value is $1/P(\mathcal{S})$, from which we derive an estimate for the number of bits E scanned at the beginning of a block until synchronization as:

$$E = \frac{W}{P(\mathcal{S})}. \tag{2}$$

In the experimental section below, we bring examples of this expected value and of actual empirical results.

4. Experimental results

We now report on some experiments with the paralel algorithm on various files. The first set consisted of textual files in different languages: the Bible (King James Version) in English, the *Dictionnaire philosophique* of Voltaire in French and the Bible in Hebrew. These files were Huffman encoded according to their individual characters. In the second set, the same files were encoded as a sequence of bigrams, yielding much larger alphabets. In the third set, we took three files of the Calgary corpus. Canonical Huffman codes were used throughout, which indeed gave noticeably faster synchronization than the other Huffman codes we tried.

Table 1 summarizes the results. The first columns give values calculated from the files themselves: the size n of the alphabet used to compress the file, the average codeword length W, the synchronization probability $P(\mathcal{S})$ of eqn. (1) and the expected number of processed bits until synchronization, E, of eqn. (2). The following columns contain values that have been empirically measured: first the average and maximum number of bits until synchronization. The numbers reported for the synchronization correspond to a block size of 512 bytes (4096 bits). The final two columns give the time, in seconds, of decoding the files sequentially and in parallel with 4 processors, using as block-size a quarter of the file-size. The time measurements were taken on a Sun 450 with four UltraSPARC–II 248 MHz processors.

Other block sizes were also checked, but essentially the same behaviour was obtained for 700, 900 and 1024 bytes. This shows that the block sizes were large enough to support the assumption that the position of a block boundary occurs at random.

As can be seen, the expected values of the number of bits to be processed until synchronization at the beginning of a block fit generally well the average of the actual values measured. As expected, synchronization is obtained faster for distributions with small average codeword length, in our examples typically in less than 100 bits, which is only 0.25% of the size of the block. But even for the larger alphabets only a few tens of bytes were needed, which is reasonable since the size of the block can be chosen larger than in our tests. For the processing time, we obviously did not expect a reduction to a quarter of the sequential speed, since beside the overlap of the blocks to be processed, there is also some overhead for the parallelization. The values we

	n	W	$P(\mathcal{S})$	E	# bits till sync avg	# bits till sync max	Decode time sequential	Decode time parallel
English	63	4.42	0.42	9.4	8.1	63	11.75	3.40
French	56	4.50	0.43	10.6	7.9	36	1.44	0.39
Hebrew	26	4.07	0.40	10.2	9.8	98	3.53	1.21
English–2	1121	8.08	0.17	47.6	72.3	675	11.48	3.28
French–2	713	7.86	0.20	39.2	37.2	257	1.73	0.54
Hebrew–2	562	7.69	0.22	35.7	33.6	240	3.99	1.40
obj1	256	6.04	0.25	24.0	14.0	112	0.05	0.02
paper1	95	5.01	0.34	15.0	10.6	39	0.10	0.05
bib	81	5.24	0.31	16.8	13.5	68	0.25	0.11

TABLE 1: *Calculated and measured values for parallel decoding*

obtained for 4 processors were typically around one third of the sequential decoding time.

5. Application to JPEG

We shortly sketch here an application to lossy image compression. In the final encoding phase of standard JPEG [17, 12], Huffman coding may be used to compress the DC and AC coefficients. The above idea, with a few adaptations, can thus be applied to decompress such JPEG files in parallel, which can yield faster reconstruction of the image when several processors are available.

A JPEG encoded file consists of codewords belonging to several Huffman codes, intermixed with strings representing numbers. When decoding is started at the beginning of a block, it is not clear which Huffman tree should be used, if at all. We shall use the tree for the AC values since they are more frequent. Once we get synchronization, we still don't know where the decoded block is to be placed. We can then choose an estimated location, at about $(i-1)/n$ of the decoded image for the output of processor i when n processors are available. Only when processor $i-1$ finishes its block will the correct position of the output of processor i be known, so blocks that have been temporarily displayed at the estimated location will probably have to be relocated.

As to the DC values, they are not encoded themselves, but rather as the difference between the current value and that of the previous block. When decoding does not start at the beginning of the file, the exact DC for the current blocks are not known. One can then assume some arbitrary basis value for DC (for example, the middle value zero) to enable the decoding of the chain of DC values within a block. A wrong guess may results in a biased image, which can be too bright or too dark for greyscale pictures, or if the change was in the luminancy component; a change in the chrominancy component of color pictures may turn the image too reddish or bluish.

This is still better than not seeing this part of the image at all. Once processor $i-1$ gets to block i, this bias will be corrected.

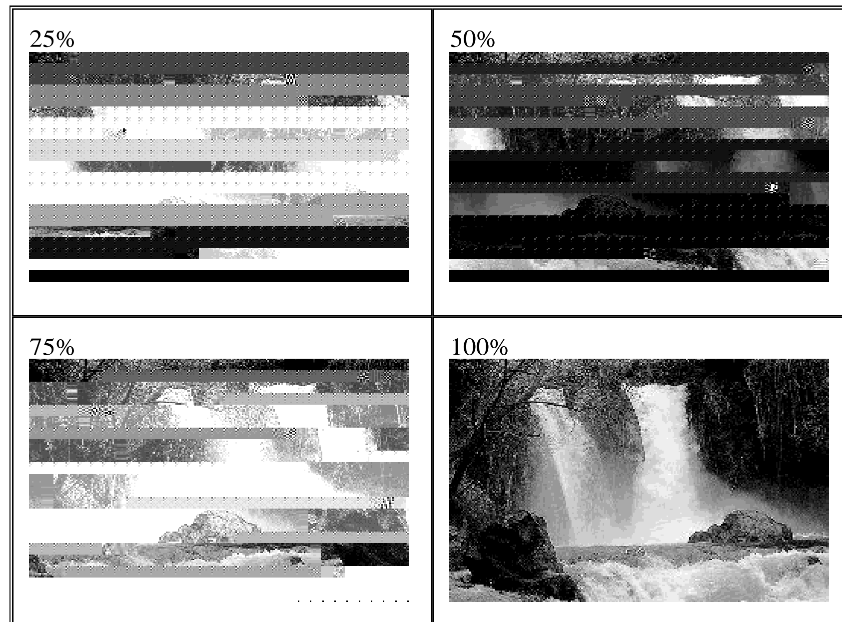

FIGURE 3: *Parallel decoding of JPEG encoded image*

Figure 3 brings an example greyscale picture, decoded by 8 processors, after 25%, 50%, 75% and the full 100% have been decoded. As can be seen, even though the partial information is not always located in its final destination, one can nevertheless recognize small sub-parts, so that the method described herein may also be useful for accelerating JPEG decoding.

References

[1] BOOKSTEIN A., KLEIN S.T., ZIFF D.A., A systematic approach to compressing a full text retrieval system, *Information Processing & Management* **28** (1992) 795–806.

[2] BOOKSTEIN A., KLEIN S.T., Is Huffman coding dead?, *Computing* **50** (1993) 279–296.

[3] 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 (1995) 172–181.

[4] 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 (1992) 52–61.

[5] FERGUSON T.J., RABINOWITZ J.H., Self-synchronizing Huffman codes, *IEEE Trans. on Inf. Th.* **IT–30** (1984) 687–693.

[6] FRAENKEL A.S., MOR M., PERL Y., Is text compression by prefixes and suffixes practical? *Acta Informatica* **20** (1983) 371–389.

[7] FRAENKEL A.S., KLEIN S.T., Bidirectional Huffman Coding, *The Computer Journal* **33** (1990) 296–307.

[8] GILBERT E.N., MOORE E.F., Variable-length binary encodings, *The Bell System Technical Journal* **38** (1959) 933–968.

[9] GONZALEZ SMITH M.E., STORER J.A., Parallel Algorithms for Data Compression, *Journal of the ACM* **32**(2) (1985) 344–373.

[10] HIRSCHBERG D.S., STAUFFER L.M., Parsing Algorithms for Dictionary Compression on the PRAM, *Proc. Data Compression Conference DCC–94*, Snowbird, Utah (1994) 136–145.

[11] HUFFMAN D., A method for the Construction of Minimum Redundancy Codes, *Proc. of the IRE* **40** (1952) 1098–1101.

[12] Information Technology - Digital Compression and Coding of Continuous–Tone Still Images Requirements and Guidelines. *International Standard ISO/IEC 10918-1* (1993).

[13] KLEIN S.T., Space and time-efficient decoding with canonical Huffman trees, *Proc. 8th Symp. on Combinatorial Pattern Matching,* Aarhus, Denmark, *LNCS* **1264**, Springer Verlag, Berlin (1997) 65–75.

[14] LAM W-M., KULKARNI S.R., Extended Synchronizing Codewords for Binary Prefix Codes, *IEEE Trans. Information Theory* **IT–42** (1996) 984–987.

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

[16] NAVARRO G., RAFFINOT M., A general practical approach to pattern matching over Ziv-Lempel compressed text, *Proc. 10th Symp. on Combinatorial Pattern Matching,* Warwick, UK, *LNCS* **1645**, Springer Verlag, Berlin (1999) 14–36.

[17] WALLACE G.K., The JPEG Still Picture Compression Standard, *Communication of the ACM* **34** (1991) 30–44.

[18] WITTEN I.H., MOFFAT A., BELL T.C., *Managing Gigabytes, Compressing and Indexing Documents and Images*, International Thomson Publishing, London (1994).

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

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