The Computer Journal, 46(5), © British Computer Society 2003; all rights reserved

Parallel Huffman Decoding with Applications to JPEG Files*

S. T. KLEIN[1] AND Y. WISEMAN[1,2]

[1]*Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel*
[2]*Computer Science Department, Jerusalem College of Technology, Jerusalem 91160, Israel*
Email: tomi@cs.biu.ac.il, wiseman@cs.biu.ac.il

A simple parallel algorithm for decoding a Huffman encoded file is presented, exploiting the tendency of Huffman codes to resynchronize quickly, i.e. recovering after possible decoding errors, in most cases. The average number of bits that have to be processed until synchronization is analyzed and shows good agreement with empirical data. As Huffman coding is also a part of the JPEG image compression standard, the suggested algorithm is then adapted to the parallel decoding of JPEG files.

Received 22 May 2002; revised 11 February 2003

1. INTRODUCTION

Huffman coding is still one of the popular compression techniques and is widely used by itself [1, 2] or in connection with other methods such as `gzip` and JPEG. Huffman's original method [3] 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 [4, 5], are the preferred choice. There are, however, situations in which a static method is required, such as for large static information retrieval systems [6] or when searching for patterns directly in a compressed text [7]. Another area of application is when more than one processor is available, in which case a static compression scheme, i.e. one that does not change the codewords dynamically during processing, may allow the decoding of several data pieces in parallel.

It is on this last point that we concentrate in this paper. We 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 the case of an error. This will then be extended to deal with the parallel decoding of JPEG files. JPEG is a widely used standard image compression technique [8] and the last phase of its baseline implementation includes Huffman coding. We shall assume a basic knowledge of Huffman's algorithm and of the properties of Huffman codes, in particular of canonical Huffman codes, which can be found in many good textbooks, for example [1].

Previous work on parallelizing compression includes [9, 10, 11], which deal with LZ compression and [12]. A parallel method for the construction of Huffman trees can be found in [13] and parallel encoding has been addressed in [14]. In fact, parallel static Huffman *encoding* is quite simple: the input text can be split among the available processors, each of which would collect statistics in its assigned block about the frequency of occurrence of the various characters. The counters of the different processors then have to be added and a global Huffman tree will be constructed, although no parallelization can be used for this step. The actual encoding can then again be performed in parallel for each block independently. Using such parallel encoding assumes alignment at block boundaries, so that some bits may be lost at the end of each block.

Our focus, however, is on *decompression*, because it may be more important than compression in some cases. For instance, in information retrieval applications, 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.

The organization of this paper is as follows: in the next section we review the main problem faced by parallel decompression, namely synchronization. Section 3 presents the algorithm and Section 4 presents some experimental results relating to general Huffman encoded files. Section 5 then deals in detail with JPEG decoding and explains how the general idea of the parallel decoding can be adapted to fit the specific characteristics of a JPEG file.

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 lengths, a block-boundary does not

*This is an extended version of a paper that appeared earlier in Klein, S. T. and Wiseman, Y. (2000) *Proc. Data Compression Conf. DCC'00*, Snowbird, Utah, pp. 383–392. IEEE Computer Society Press, Los Alamitos, CA.

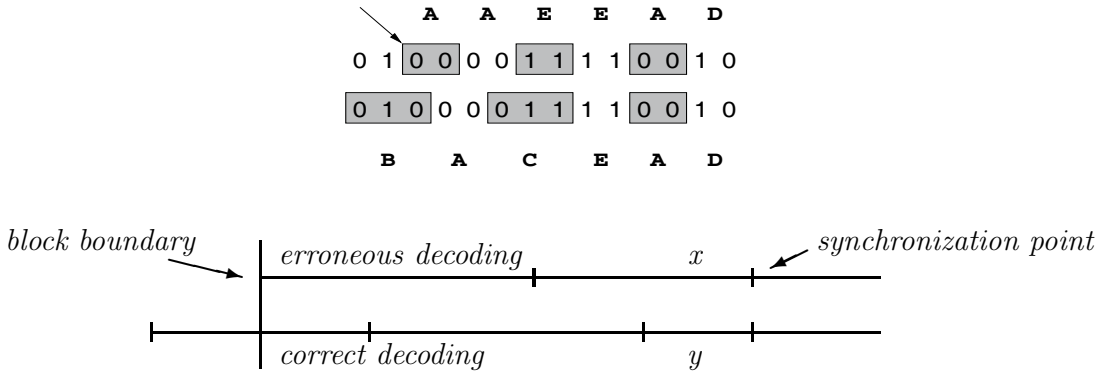

FIGURE 1. Schematic representation of parallel decoding.

necessarily coincide with a codeword boundary. However, 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 (see [15] for a definition of *synchronization* in our context).

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, indicated by the arrow in the upper part of Figure 1. It would then decode the block as AAEEAD, the first three characters of which are erroneous.

The general situation is given in the lower part of 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 a *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 block size is not a multiple of k, all the codewords of the second block will be out of synchronization. However, in the case of a fixed length code, the block size could be chosen *a priori* as a multiple of the codeword length. Moreover, fixed length codes are only optimal, from the compression point of view, 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, similar 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 [15]; they are called *affix* codes in [16]. 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 [17]. 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 [18] did not yield affix codes.

For certain distributions, a Huffman code may be constructed that includes synchronizing codewords or sequences [19, 20]. 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 [19] 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 simplest approach to allow parallel decoding is to decide in advance on the sizes of the blocks and force alignment at block boundaries by inserting padding bits at the end of the blocks. Note, however, that padding cannot always be done simply by inserting zeroes or random bits, since such a sequence of padding bits might turn out to be 'decodable', yielding erroneous decoding. The padding should thus consist of a proper prefix of one of the codewords.

An easy way to implement this for a fixed block size b would be as follows: Start with a Huffman encoded string T and repeat until T is empty: remove from T a prefix of size b—this prefix constitutes the following block B; if the last codeword w did not fit in its entirety in B, prepend the prefix of w which is a suffix of B in front of T. The average number per block of added bits will be about half of the average codeword length, which might be negligible if the block size is large enough.

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

If   EOB is an eoc or EOB is EOF (last block)   STOP
else                    // overflow to next block
{       i ⟵ i + 1
        p ⟵ head_i
        repeat
        {       decode to next eoc
                if   this eoc is EOB   STOP
                if   EOB was passed
                {       i ⟵ i + 1       p ⟵ head_i      }
                else
                {       j ⟵ index of eoc in block i
                        c ⟵ corresponding decoded character
                        if L_i[p] not yet defined, wait until defined
                        while   L_i[p].index < j
                        {       p_old ⟵ p
                                p ⟵ p.next
                                remove L_i[p_old] from L_i
                                if L_i[p] not yet defined, wait until defined
                        }
                        if   L_i[p].index = j   STOP
                        else   insert (j, c) in front of L_i[p] in L_i
                }
        }
}
```

FIGURE 2. Decoding algorithm for processor i.

Alternatively, instead of repeating a prefix of the last codeword for each block, one could record the length ℓ of this prefix in a separate table. Processor i would then start its work by reading a suffix of length ℓ from block $i-1$ before turning to its own block i. The overhead per block in this case is log(maximum codeword length), which again may be very small relative to a large block. It should, however, be noted that assuming a lower bound to the size of a block is equivalent, as the size of the input file is given, to an upper bound on the number of processors. So if many processors are available and the input file is not very large, either we agree not to take advantage of all the given processing power or one has to deal also with smaller blocks. In the latter case, increasing each block with padding bits or adding an integer ℓ to each block might no longer be a negligible overhead.

In any case, even if the overhead is very small, the main disadvantage is that the number of processors has to be fixed in advance, at encoding time. The algorithm we suggest below does not alter the original Huffman encoded file and has therefore no overhead. Moreover, any number of processors can be accommodated, the exact number is only needed at decoding time and may even change from one decoding to another.

3.1. Description of the suggested algorithm

The basic idea of the parallel decoding algorithm is to let 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.

The formal parallel decoding algorithm for processor i is given in Figure 2. Processor i maintains a linked list L_i of pairs $(index, char)$, which is also accessible to processors j, for $j < i$, and records the indices within block i, of the last bit of each codeword, as well as the corresponding decoded characters. Denote by $L_i[p]$ the element of L_i pointed to by p and by $L_i[p].index$ and $L_i[p].char$ the index and character fields within $L_i[p]$, respectively. In general, the first few elements of L_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. The list L_i also serves as an indicator for processor i to stop: as soon as an index value is detected that is equal to one of the index values stored in L_i by an earlier processor, synchronization has been achieved. The final decoded sequence can then be obtained by accessing in parallel the lists $L_i[p].char$.

We use the abbreviations EOB for end of block, EOF for end of file and eoc for end of codeword. A pointer $head_i$ points to the head of L_i and if p points to an element of L_i, $p.next$ points to its successor in the linked list. The wait statements have been added to allow correct processing even in the (quite unlikely) case that processor $i-1$ finishes work on block $i-1$, moves to block i and gets to the end of some codewords there, before processor i has reached these codewords in its own block.

To show correctness, we use an inductive argument: note that getting a correct decoding in block $i+1$ is based on the assumption that the last codewords in block i have been correctly decoded by processor i. If this assumption 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, also continue working 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 behavior seems to be extremely rare, as in most cases the synchronization points are found quickly, long before the end of the block.

3.2. Analysis

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 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. However, 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 accommodate a very large number of consecutive codewords, we conclude that our assumption can be justified.

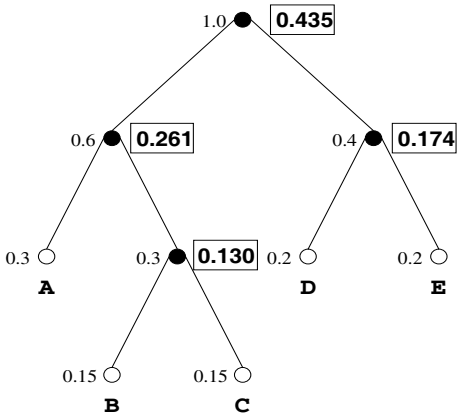

FIGURE 3. Probabilities $P(x)$ for an example Huffman tree.

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 randomly pick bit positions which will 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 $\sum_{x \in \mathcal{I}} P(x) = 1$. As an example, refer again to the simple Huffman code $\{00, 010, 011, 10, 11\}$ mentioned earlier and assume that the characters A, B, C, D, E appear with probabilities 0.3, 0.15, 0.15, 0.2 and 0.2, respectively, yielding an average W of 2.3. Figure 3 shows the corresponding tree, with each node having its associated probability to its left. The probabilities $P(x)$ appear in boxes to the right of the internal (black) nodes.

For $x \in \mathcal{I}$ and $y \in \mathcal{L}_x$, define

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

that is $Q(x, y) = 1$ if and only if, in the case where a codeword has been cut by a block boundary, synchronization is re-established 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 the example in Figure 3, $Q(x, y)$ is 1 if x is the root or if x is the internal node corresponding to the prefix 0 and y is one of the leaves corresponding to B or C.

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:

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

However, $P(\mathcal{S} \mid \text{cut-point is at } x)$ 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

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

We therefore conclude that the probability $P(\mathcal{S})$ 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 [18]. Such a shape tends to favor reoccurring 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. However, 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

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

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 parallel 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 gave noticeably faster synchronization than the other Huffman codes we tried.

Table 1 summarizes the results. The first columns give the 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 Equation (1) and the expected number of processed bits until synchronization, E, of Equation (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 four processors, using as block-size a quarter of the file-size. Standard Huffman decoding was used for both the sequential and parallel programs. More sophisticated decoding procedures exist [18, 21], but we did not want to bias the relative gain in speed due to parallelization. The time measurements were taken on a Sun 450 with four UltraSPARC–II 248 MHz processors. The model we used is the shared memory introduced in Solaris 2.6 [22], protected by the standard Unix System V semaphores and allocated according to the Slab Allocator [23].

Other block sizes were also checked, but essentially the same behavior 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. For such large blocks, the overhead of forcing alignment by padding would also be very low, less than 0.1% for all our examples. However, if blocks as small as 40 bytes—still much more than needed to

TABLE 1. Calculated and measured values for parallel decoding.

					Number of bits until synchronization		Decode time	
	n	W	$P(\mathcal{S})$	E	Average	Maximum	Sequential	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

get synchronization—are permitted, the padding overhead would increase up to 2%.

As can be seen, the expected values of the number of bits to be processed until synchronization at the beginning of a block generally fit the average of the actual values measured well. 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. However, 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 besides the overlap of the blocks to be processed, there is also some overhead for the parallelization. The values we obtained for four processors were typically around one-third of the sequential decoding time.

5. APPLICATION TO JPEG

5.1. Baseline JPEG

We start with an overview of the essentials of JPEG needed to understand the details of the parallel decoding. JPEG [24] is a lossy image compression method. In a first step, the picture is split into a sequence of blocks of size 8×8 pixels. Each block is then compressed by the following sequence of transformations.

1. Applying a *discrete cosine transform* (DCT) [25] to the set of 64 values of the pixels in the block.
2. Applying *quantization* to the DCT coefficients, thereby producing a set of 64 smaller integers. This step causes a loss of information but makes the data more compressible.
3. Applying an *entropy encoder* to the quantized DCT coefficients. Baseline JPEG uses Huffman coding in this step, but the JPEG standard also specifies arithmetic coding [26] as a possible alternative.

The decompression process just reverses the actions and their order. It first applies Huffman decoding, then dequantizes the coefficients and finally uses an inverse DCT to obtain the original set of values. Because of the quantization step, the reconstructed set includes only approximated values.

The coefficient in position $(0, 0)$ (left upper corner) is called the *DC coefficient* and the 63 remaining values are called the *AC coefficients*. In principle, the DC coefficient should store a measure of the average of the 64 pixel values of the given block, but since there is usually a strong correlation between the DC coefficients of adjacent blocks, what is actually stored is the difference between the average in this block and the average in the previous one.

Baseline JPEG uses two different Huffman trees to encode the data. The first encodes the lengths in bits (1 to 11) of the binary representations of the values in the DC fields. The second tree encodes information about the sequence of AC coefficients. As many of them are zero, and most of the non-zero values are often concentrated in the upper left part of the 8×8 block, the AC coefficients are scanned in a zig-zag order, processing elements on a diagonal close to the upper left corner before those on such diagonals further away from that corner, that is the order is given by $(0, 1)$, $(1, 0)$, $(2, 0)$, $(1, 1)$, $(0, 2)$, $(0, 3)$, $(1, 2)$, etc. The second Huffman tree encodes pairs of the form (n, ℓ), where n (limited to 0 to 15) is the number of elements that are zero, preceding a non-zero element in the given order and ℓ is the length in bits (1 to 10) of the binary representation of the non-zero quantized AC value. The second tree also includes codewords for EOB, which is used when no non-zero elements are left in the scanning order and for a sequence of 16 consecutive zeros in the AC sequence (ZRL), necessary to encode zero-runs that are longer than 15. The Huffman trees used in baseline JPEG are static and can be found in [8].

Each 8×8 block is then encoded by an alternating sequence of Huffman codewords and binary integers (except that the codeword for ZRL is not followed by any integer), the first codeword belonging to the first tree and relating to the DC value, the other codewords encoding the (n, ℓ) pairs for the AC values, with the last codeword in each block representing EOB. Figure 4a brings an example block of quantized values, with the DC value in boldface in the upper left corner. The upper part of Figure 4b shows the encoding

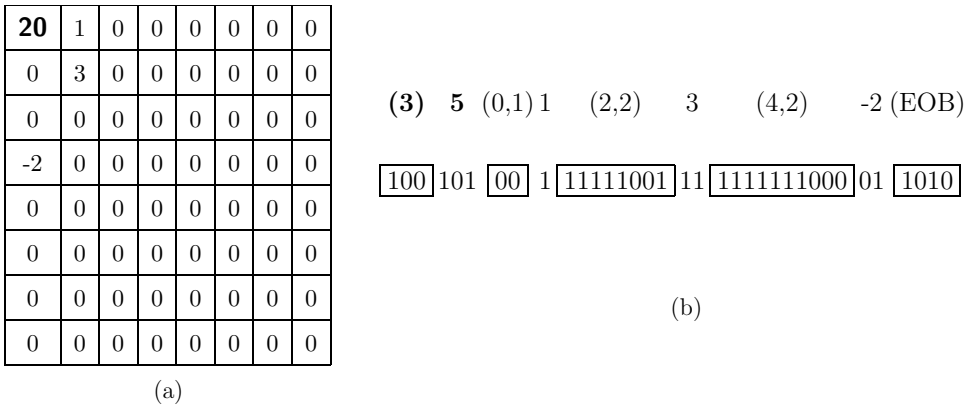

FIGURE 4. Example of a JPEG block (a) and its encoding (b).

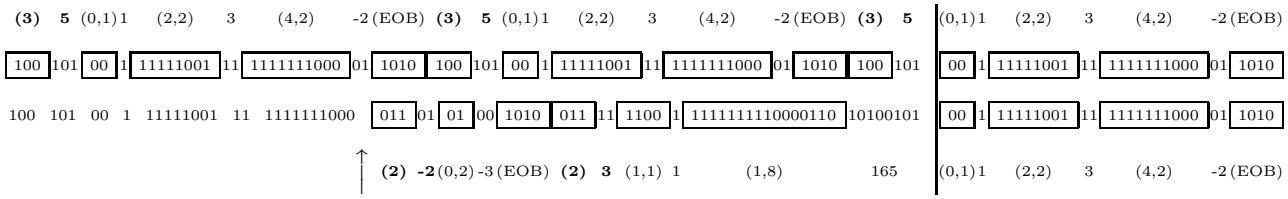

FIGURE 5. Example of wrong decoding and synchronization.

of this block, with Huffman encoded elements appearing in parentheses; the binary translation of the encoding, with framed Huffman codewords, is shown underneath.

Turning now to the problem of parallel decoding, the above idea, with a few adaptations, can be applied to decompress Huffman based JPEG files in parallel, which can yield faster reconstruction of the image when several processors are available. One possible approach to adjust JPEG to a parallel scheme is to change the basic JPEG scheme to a more adequate format, as suggested in [27, 28]. Our goal, however, is a method capable of decoding in parallel without changing the standard.

As was done above for general Huffman encoded files, we start by splitting the image into several slices and assigning different processors, each working on a different slice of the image. The synchronization problems mentioned in Section 2 also appear here and are even more severe. Not only does the beginning of the block to be decoded by the current processor not necessarily coincide with the beginning of a Huffman codeword, but even if it does synchronization is not guaranteed. The following different cases may occur—the block boundary could be located:

- within the codeword representing the length of the DC coefficient;
- at the beginning or within the stored DC value;
- at the beginning or within a codeword representing an (n, ℓ) pair used for the AC coefficients;
- at the beginning or within a stored AC value.

Only if the block happens to start with a codeword for the length of the stored DC value will the block be correctly decoded.

To illustrate this problem, assume that the block used in the example in Figure 4 appears consecutively three times in the given file. Suppose then that a new processor starts working six bits before the end of the first block, which corresponds to the beginning of the binary encoding of the AC value -2. The processor would try to recognize a Huffman codeword representing the length of a DC value and would thus erroneously interpret the next *three* bits 011 as representing the length 2, implying further errors. Figure 5 shows, in its upper part, the correct decoding and, in its lower part, the decoding obtained when starting at the sixth bit before the end of the first block, as indicated by the arrow. As can be seen, the first few decoded elements are wrong, but soon a synchronization point, indicated by the vertical bar, is found after which the decoded elements are correct.

In this example, the first decoded block is completely wrong and the second includes at its end some correctly decoded AC coefficients, which, however, are useless because of their wrong positions within the block. Only after the second EOB will the correct decoding resume. In general, correct decoding and not just synchronization will only be achieved after a correct EOB codeword is detected. Note that two different errors may occur: a true occurrence of EOB (1010 in our example) may be overlooked, as the first and second EOB in the upper part of Figure 5, or an EOB may be detected even though there is none in the true decoding, as in the case of the first EOB of the lower part of Figure 5. The correct EOB after which decoding is correct is the first EOB found after synchronization.

The parallel decoding algorithm for baseline JPEG is similar to the general algorithm of Section 3, with the following additions.

(a) (b)

FIGURE 6. Example of the parallel decoding of a greyscale image: (a) parallel decoding with four processors; (b) original image.

5.1.1. Invalid codewords

Any Huffman code is complete, in the sense that any binary string can be 'decoded' as if it were some Huffman encoding, so that errors in the binary file will stay undetected unless the end of the file is reached within a codeword. In the particular case of JPEG, errors may be detected in certain circumstances: keeping track of the number of AC elements by summing the n fields of the (n, ℓ) pairs and adding the number of non-zero coefficients, if this number exceeds 63, there must obviously have been an error. In addition, the particular Huffman trees used (see [8]) are not complete and, in fact, certain codewords are missing (for example 111111111 in the first Huffman tree, used for the DC coefficients). The appearance of such an invalid codeword is therefore a sign that some error has occurred. As it makes no sense to display a block which is obviously incorrect, an empty block will be substituted. The error will be fixed when the decoder of the previous block will overflow into the current block.

5.1.2. Positioning of the image

Another factor applying to the decoding of JPEG files is the possibility to display partial data while decoding, even if the correct location is yet unknown. As the compressed data has variable length, the location of each block in the image can not be known accurately. The algorithm will then choose an estimated location, at about $(i-1)/k$ of the decoded image for the output of processor i if k 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.

The wrong positioning of a decoded block may cause the cutting of a line of blocks into two parts: the left part may appear at the right end of a line of blocks in the image, whereas the right part will be at the left border of the following line. For many images this will result in a discontinuity which is often easily detectable by a human eye, as for example in the lower part of Figure 6a above. A straightforward idea would then be to try to detect if such discontinuities (which are equivalent to high DC values, since these store differences from preceding blocks) happen to reoccur at the same position in consecutive lines, suggesting that this position should be moved to the edge of the image. However, such a rule of thumb may fail, either in the case when blocks at the left or right edges of the image are similar, or when there is a true discontinuity in the given position. Moreover, the algorithm would be more time consuming, which might cancel a part of the gain in speed due to the parallel processing.

5.1.3. Adjusting the color

As mentioned, the DC values 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 result in a biased image, which can be too bright or too dark for greyscale pictures, if the change was in the luminance component; a change in the chrominance 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 all the preceding processors get to the blocks following the one they have been assigned to, this bias will be corrected. This means, however, that to get a correct decoding of the picture, we actually have to process it sequentially. The advantage of the parallel decoding reduces in this case to the ability of getting quickly some partial information in the form of a biased picture, that will subsequently be rectified.

Note that the new standard JPEG–2000 [29] has built-in synchronization codewords which make the synchronization faster. Obviously, this will improve the performance of the parallel decoding application.

Figure 6 is an example of the decoding of a greyscale picture, decoded by four processors. Figure 6a shows the reconstructed image after having processed all the blocks, but before letting the processors overflow into the adjacent blocks to correct the wrong positioning and luminance. Figure 6b brings then the corrected picture.

(a) (b)

FIGURE 7. Example of the parallel decoding of a color image: (a) parallel decoding with four processors; (b) original image.

TABLE 2. Statistics on JPEG decoding.

	Average size of 8×8 block	Number of bits until synchronization		Size of picture (number of 8×8 blocks)
		Average	Maximum	
Greyscale image	144.1	74.9	1815	63×43
Color image	339.3	150.1	1853	20×29

5.2. Other JPEG formats

The JPEG standard [24] has some other formats for encoding images. In several *progressive modes*, the scanning order is altered. In one of the variants, for instance, using 8 bits for each encoded coefficient, the $64 \times 8 \times k$ bits of the k blocks constituting an image are rearranged, clustering the $64k$ most significant bits together, followed by the $64k$ bits in position 2 of each coefficient, etc. The resulting sequence is then Huffman encoded. This mode of transmission has the advantage of permitting a sequence of approximations of the image to be obtained before all of the data has been read. The first approximation will be blurred, but the consecutive ones will become progressively clearer. Unfortunately, this and similar rearrangements are not suitable for our parallel decoding: even if synchronization is regained, we still do not know to which of the k blocks the decoded values have to be assigned, nor do we have any information about the index of the bit currently processed.

JPEG also deals with pictures encoded by several components, such as color images. There are several methods for splitting a color pixel into components [30, 31]. The common standards are RGB [32] and YUV [33]. A color picture is not decomposed into three independent images in each of the color shades, but rather for each block, the three components (RGB or YUV) are encoded consecutively. In the parallel decoding process, the current color shade (e.g. R, G or B) after synchronization can only be guessed and will be corrected if necessary when all the preceding processors overflow into the next image slices. Until this correction is performed, the color shades may be incorrect. Figure 7 is an example of the decoding of a color image by four processors, similar to Figure 6.

Table 2 displays some statistics about the parallel JPEG decoding. Note that because the encoded file is not just a sequence of Huffman codewords, but also includes various integer encodings, the relevant number is not the average size of a Huffman codeword but the average size of an encoded block of 8×8 pixels, which is given, in bits, in the first column. The number of bits till synchronization is thus also measured up to the beginning of the following correctly decoded 8×8 block. For the color image, each block consists, in fact, of three independent ones (for Y, U and V). We see that synchronization is achieved on average after about half a block, which, given the size of the pictures (last column), is hardly noticeable. Timing figures have not been included: as mentioned above, full decoding including relocation and color adjustment might force a sequential

scan. On the other hand, if we measure the decoding time only until each processor finishes its own block, the processors can work independently of each other, so the time is reduced by a factor of n if n processors are available.

6. CONCLUDING REMARKS

Parallelization of the decoding process of a Huffman compressed file seems to be an easy task if one is willing to change the original compressed file slightly, forcing codeword alignment at block boundaries. The incurred overhead will often be negligible. The new algorithm does not alter the encoded file at all, using the well-known property of Huffman codes to resynchronize quickly after errors. We have analyzed the average length of the segment until synchronization is achieved and compared the theoretically expected values with experimental results on real-life data, showing generally good agreement.

The basic idea has then been extended to deal with the parallel decoding of JPEG encoded images, since Huffman coding is a part of the JPEG scheme. Decoding is more involved in this case and to get the correct reconstructed picture, it may at times take as long as for the sequential procedure. The benefit of using parallel decoding reduces then to the ability of getting faster partial visual information.

In fact, the technique of letting the processors overflow to neighboring blocks might have applications beyond those of parallel decompression. Any divide and conquer scheme splits its input into several independent parts, which are then processed (in parallel or sequentially) individually and whose results are then somehow combined. Usually, the division points between the parts are well defined. For certain applications, however, it might not be clear where to choose the boundaries of the partition. In such cases, permitting a certain overlap between the parts similar to that of our parallel Huffman decoding, might possibly yield easier ways for the recombination of the results.

ACKNOWLEDGEMENT

The authors wish to thank the three anonymous referees for their helpful comments.

REFERENCES

[1] Witten, I. H., Moffat, A. and Bell, T. C. (1994) *Managing Gigabytes, Compressing and Indexing Documents and Images*. International Thomson Publishing, London.

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

[3] Huffman, D. (1952) A method for the construction of minimum redundancy codes. In *Proc. IRE*, Kansas City, 9 September 1952, vol. 40, pp. 1098–1101. The Institute of Electronics and Radio Engineers, London.

[4] Ziv, J. and Lempel, A. (1977) A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theory*, **23**, 337–343.

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

[6] Bookstein, A., Klein, S. T. and Ziff, D. A. (1992) A systematic approach to compressing a full text retrieval system. *Inform. Process. Management*, **28**, 795–806.

[7] Navarro, G. and Raffinot, M. (1999) A general practical approach to pattern matching over Ziv-Lempel compressed text. In *Proc. 10th Symp. on Combinatorial Pattern Matching*, Warwick, UK, 22–24 July 1999, *Lecture Notes in Computer Science*, **1645**, pp. 14–36. Springer, Berlin.

[8] Wallace, G. K. (1991) The JPEG still picture compression standard. *Commun. ACM*, **34**, 30–44.

[9] De Agostino, S. and Storer, J. A. (1995) Near optimal compression with respect to a static dictionary on a practical massively parallel architecture. In *Proc. Data Compression Conf. DCC–95*, Snowbird, Utah, 28–30 March 1995, pp. 172–181. IEEE Computer Society.

[10] De Agostino, S. and Storer, J. A. (1992) Parallel algorithms for optimal compression using dictionaries with the prefix property. In *Proc. Data Compression Conf. DCC–92*, Snowbird, Utah, 24–27 March 1992, pp. 52–61. IEEE Computer Society.

[11] Gonzalez Smith, M. E. and Storer, J. A. (1985) Parallel algorithms for data compression. *J. ACM*, **32**(2), 344–373.

[12] Hirschberg, D. S. and Stauffer, L. M. (1994) Parsing algorithms for dictionary compression on the PRAM. In *Proc. Data Compression Conf. DCC–94*, Snowbird, Utah, 29–31 March 1994, pp. 136–145. IEEE Computer Society.

[13] Lawrence, L. L. and Przytycka, T. M. (1995) Constructing Huffman Trees in parallel. *SIAM J. Comput.*, **24**(6), 1163–1169.

[14] Howard, P. G. and Vitter, J. S. (1992) Parallel lossless image compression using Huffman and aithmetic coding. In *Proc. Data Compression Conf. DCC–92*, Snowbird, Utah, 24–27 March 1992, pp. 299–308. IEEE Computer Society.

[15] Gilbert, E. N. and Moore, E. F. (1959) Variable-length binary encodings. *Bell Syst. Tech. J.*, **38**, 933–968.

[16] Fraenkel, A. S., Mor, M. and Perl, Y. (1983) Is text compression by prefixes and suffixes practical? *Acta Inform.*, **20**, 371–389.

[17] Fraenkel, A. S. and Klein, S. T. (1990) Bidirectional Huffman coding. *Comput. J.*, **33**, 296–307.

[18] Klein, S. T. (2000) Skeleton trees for the efficient decoding of Huffman encoded texts. *Kluwer J. Inform. Retrieval*, **3**, 7–23.

[19] Ferguson, T. J. and Rabinowitz, J. H. (1984) Self-synchronizing Huffman codes. *IEEE Trans. Inform. Theory*, **30**, 687–693.

[20] Lam, W. M. and Kulkarni, S. R. (1996) Extended synchronizing codewords for binary prefix codes. *IEEE Trans. Inform. Theory*, **42**, 984–987.

[21] Brodnik, A. and Carlsson, S. (1998) *Sublinear Decoding of Huffman Codes Almost in Place*. Technical Report 36/600, IMFM, Ljubljana, Slovenia.

[22] Vahalia, U. (1996) *UNIX Internals—The New Frontiers*. Prentice-Hall, Englewood Cliffs, NJ.

[23] Bonwick, J. (1994) The slab allocator: an object-caching kernel memory allocator. In *Proc. Summer 1994 USENIX Tech. Conf.*, Boston, Massachusetts, 6–10 June 1994, pp. 87–98. USENIX, Berkeley, CA, 94710, USA.

[24] ISO/IEC 10918-1 (1993) Information technology—digital compression and coding of continuous–tone still images requirements and guidelines. *International Standard ISO/IEC*, Geneva, Switzerland.

[25] Rao, K. R. and Yip, P. (1990) *Discrete Cosine Transform Algorithms, Advantages, Applications*. Academic Press, London.

[26] Witten, I. H., Neal, R. M. and Cleary, J. G. (1987) Arithmetic coding for data compression. *Comm. ACM*, **30**, 520–540.

[27] Yun, D. Y. Y. and Chen, C. (1996) ESS Project, Annual Report FY96—Applications, NASA, USA Government, Greenbelt, MD.

[28] Yun, D. Y. Y. and Chen, C. (1997) ESS Project, Annual Report FY97—Applications, NASA, USA Government, Greenbelt, MD.

[29] Marcellin, M. W., Gormish, M. J., Bilgin, A. and Boliek, M. P. (2000) An overview of JPEG-2000. In *Proc. Data Compression Conf. DCC-2000*, Snowbird, Utah, 28–30 March 2000, pp. 523–541. IEEE Computer Society, New Jersey.

[30] Hearn, D. and Baker, M. P. (1986) *Computer Graphics*. Prentice-Hall, Englewood Cliffs, NJ.

[31] Jain, A. K. (1986) *Fundamentals of Digital Image Processing*. Prentice-Hall, Englewood Cliffs, NJ.

[32] Hunt, R. W. G. (1952) *The Reproduction of Colour*. Morgan, Keene Valley, NY.

[33] Laplante, P. A. and Stoyenko, A. D. (1996) *Real Time Imaging, Theory, Techniques and Applications*. IEEE Press, New York.