Yin, Hoover H. F.; Woo, Ivy K. Y.; Lai, Russell W. F.

On the Applications and Parallelization of Multichannel Source Coding

Published in:
Proceedings of 2022 IEEE Region 10 International Conference, TENCON 2022

DOI:
10.1109/TENCON55691.2022.9977596

Published: 04/11/2022

Document Version
Peer-reviewed accepted author manuscript, also known as Final accepted manuscript or Post-print

Please cite the original version:
Yin, H. H. F., Woo, I. K. Y., & Lai, R. W. F. (2022). On the Applications and Parallelization of Multichannel Source Coding. In *Proceedings of 2022 IEEE Region 10 International Conference, TENCON 2022* (pp. 1-6). Article 9977596 (TENCON IEEE Region Ten Conference). IEEE. https://doi.org/10.1109/TENCON55691.2022.9977596

On the Applications and Parallelization of Multichannel Source Coding

Hoover H. F. Yin*, Ivy K. Y. Woo†, and Russell W. F. Lai†
*Institute of Network Coding, The Chinese University of Hong Kong
†Aalto University

***Abstract*—Each codeword in Multichannel Source Coding (MSC) naturally uses multiple channels simultaneously. Previous works only focused on the theoretical aspects of MSC. In this paper, we discuss the applications and parallelization of MSC.**

I. Introduction

Single-channel source coding is a widely studied topic where many practical compression schemes were invented. On the other hand, *multichannel source coding (MSC)*, where each codeword is a tuple of strings in different channels, is a generalization of its single-channel counterpart [1]. For simplicity, we call the single-channel version the *baseline source coding (BSC)* in this paper.

Previous works on MSC in the literature such as [1]–[3] focused on the theoretical problems induced by the extra freedom of dimensions and did not discuss any practical applications. To move from theory to practice, the main obstacle is that MSC cannot improve the optimal compression rate when all the channels have the same alphabet size [1].[1] The same channel alphabet size, e.g., binary channels, is a common setup in communications, so MSC is dubbed worthless in most applications. This fact suggests that instead of compression rates, we need to consider other advantages of MSC over BSC.

As MSC separates the codewords into multiple channels naturally, a question is that whether we can parallelize the encoding (compression) and decoding (decompression) processes with ease. For BSC, various approaches such as [7]–[10] were proposed to parallelize different source codes. The basic idea of parallelization is to divide either the source file or the compressed file into multiple parts (with or without overlapping). The latter part of a file corresponds to the data of a later time in the sense of streaming, so we call the parallelization according to "time" the *temporal parallelization*. Unlike BSC, each codeword in MSC can be spread to multiple channels naturally so that more "space" can be used at the same time. In the sense of streaming, we refer this as a *spatial parallelization*, which is another dimension enabled by using more channels at once.

In this paper, we propose some real-world use cases of MSC and discuss the practical advantages of MSC, namely, *load assignment* and *faster reception rate*. To further investigate the possibilities of MSC, we discuss the advantages and disadvantages of various spatial and temporal parallelization techniques, and propose some MSC-exclusive approaches. We also propose a constrained MSC which enables more parallelization options.

[1]In theory, MSC can be extended to where the channels do not need to use the same alphabet size [3] so that a mixture of channels of different media or quantization levels can be applied. It is possible to obtain a smaller redundancy in this setting [4], but there is no known efficient way to construct an optimal code. Also, the optimal code may utilize the channels badly, say, some channels may be left unused, in order to achieve the optimality. Under this extension, some information source may not have an optimal code which has a decoding tree [5]. On the other hand, the way to adopt more than one decoding tree, e.g., the multichannel counterpart of AIFV codes [6], is still unknown.

II. Multichannel Source Coding

A. Definitions

We first recall the basics of multichannel source coding. Let $\mathcal{Z}$ be the alphabet of the information source Z, and let $\mathcal{Y}$ be the channel alphabet of size q. Let $\mathcal{Y}^*$ be the set of all finite length strings over the alphabet $\mathcal{Y}$, including the empty string.

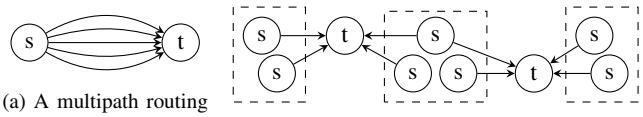
Definition 1. An n-channel q-ary *source code* for the information source Z is a mapping from $\mathcal{Z}$ to $(\mathcal{Y}^*)^n$.

That is, every source symbol is mapped to an n-tuple of strings, where each of such tuple is called a *codeword*. The i-th component (we count from 0) of the tuple is sent using the i-th channel. When we send more than one codeword, the codewords are concatenated component-wisely. Thus, the boundaries of the codewords are not longer explicit.

The *codeword length* of a codeword in channel i is the string length (the number of channel symbols) of the i-th component of the codeword. The *total codeword length* is the sum of the codeword lengths in all the channels. Similar to BSC, an optimal MSC is defined to be a code which has the smallest expected total codeword length. The redundancy of an optimal MSC is the same as that of an optimal BSC [1].

The decoding tree of MSC is a tree that every non-leaf node belongs to a *class*. A class i node is associated with the i-th channel, and each of its branch corresponds to a symbol in $\mathcal{Y}$. Every leaf node corresponds to a codeword and associates with a source symbol. To decode a codeword, we start from the root node. When we reach a class i node (including the root node), we fetch the first unread symbol from the i-th channel, which specifies the branch that we should take to reach the next node. We can decode the codeword once we reach a leaf.

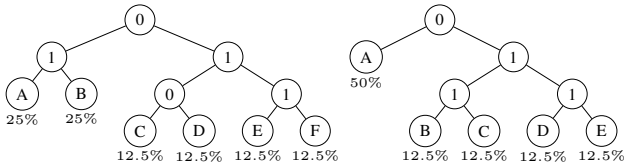
It is well-known that every single-channel prefix code has a decoding tree. However, this is not the case for multichannel prefix codes. The decoder of a general MSC may need to



(a) A multipath routing network.

(b) Clients receiving data from CDN nodes.

Fig. 1: Applications of multichannel source coding.



(a) The Huffman tree in Example 1. (b) The Huffman tree in Example 2.

Fig. 2: Decoding trees. The numbers are the classes of the nodes.

refer to the symbols of future codewords, e.g., the SPDP in [1]. On the contrary, if the MSC has a decoding tree, i.e., a *tree-decodable* MSC, then it is an instantaneous code.

Given the desired expected codeword length of each channel where the sum of these lengths equals the expected total codeword length of a given decoding tree, Yao and Yeung [1] introduced an algorithm to assign the channels (classes) to the tree such that for each channel, the expected codeword length of this channel in the resulting tree and the given desired expected codeword length of this channel is differed by at most 1 channel symbol. The complexity of this algorithm is $\mathcal{O}(nT)$, where n is the number of channels and T is the size of the tree. Although only Huffman tree is considered in [1], the algorithm can be applied to any tree-decodable MSC. If the given tree is a Huffman tree, then the resulting tree is an optimal uniquely decodable code in terms of expected total codeword length. In other words, it is sufficient to consider tree-decodable MSC in practice.

Hoever, we have some limitations when we restrict ourselves to tree-decodable MSC. First, the number of channels we can use is upper bounded by the number of non-leaf nodes. This is due to the fact that each non-leaf node is associated with one channel only. For example, by counting, a Huffman code cannot use more than $\lceil \frac{|\operatorname{Supp}(Z)|-1}{q-1} \rceil$ channels, where $\operatorname{Supp}(Z)$ is the support of Z.[2] Second, there exists a channel having expected codeword length at least 1 because the channel associated with the root node is used at least once per source symbol.

In this paper, our discussion does not impose restrictions on how the decoding tree is obtained or whether the tree is optimal. For example, we can construct a 2-channel tree [11], [12] by using the Krichevsky-Trofimov estimator [13]. Yet, we consider an optimal decoding tree for fair comparison in our experiment.

B. Applications

We now propose some real-world use cases of MSC based on the nature of MSC. First, the basic model of MSC is illustrated in Fig. 1a, which consists of a single source s, a destination t and multiple channels connecting them. A direct insight is that this model can be interpreted as a multipath routing network where each channel corresponds to a route. In other words, all the routes are aggregated into a single virtual path for transmitting the source file.

We can also view the model in Fig. 1a this way. After applying an MSC, we store each component of the compressed data in a separate file so that each channel is sending the corresponding file, where each of these files is called a *compressed file* for simplicity. We consider that each of these files can be accessed independently from different network nodes. Then, the situation becomes the destination node requests the compressed files from multiple source nodes. The main difference between distributed source coding (DSC) and MSC is that all the compressed files in MSC correspond to a single information source. On the other hand, the situation is similar to distributed storage, but MSC focuses on compression and does not have repairing ability.

This application does not require special routing techniques, so it can be applied directly in existing networks with a client program using multiple sockets. These source nodes can be part of a content delivery network (CDN) so that which servers (source nodes) to be connected can be decided remotely. This architecture is illustrated in Fig. 1b, where the three dashed boxes represent three different geographically distinct data centers. The bandwidth of the servers consumed by a client can be distributed so that each server can serve more nearby clients. Note that it is possible to request more than one compressed file from one server. That is, the channels can be virtual channels that are bundled into a real channel.

Recall that we can tune the expected codeword length of each channel for MSC. In other words, MSC has a natural **load assignment** ability: We can assign the load according to the transmission rates of the channels by controlling the expected codeword reception rate per unit time, where the client does not need extra synchronization with the server(s) during the transmission. To see this ability, we give the following example.

Example 1. Suppose $\mathcal{Z} = \{A, B, C, D, E, F\}$ and $\Pr(A) = \Pr(B) = 25\%$, $\Pr(C) = \Pr(D) = \Pr(E) = \Pr(F) = 12.5\%$. Consider a 2-channel binary MSC and we want to have the same load on both channels. The Huffman code has a expected total codeword length 2.5 so we desire the expected codeword length of each channel is 1.25. Such a decoding tree is illustrated in Fig. 2a.

As multiple channels can be utilized to send the codeword of the same source symbol, we can achieve a **faster reception rate** for each source symbol, which enables faster decoding of important information such as metadata or I-frames in

[2]Here is the proof of the statement. Let d be the minimal number of dummy leaves, and k be the number of non-leaf nodes. In each iteration of the Huffman procedure, we merge q masses into one mass and produce one non-leaf node. That is, we have $|\operatorname{Supp}(Z)| + d - (q-1)k = 1$, which gives $k = \frac{|\operatorname{Supp}(Z)|+d-1}{q-1}$. As k is an integer and d is minimal, we have $k = \lceil \frac{|\operatorname{Supp}(Z)|-1}{q-1} \rceil$. Each non-leaf node only belongs to one class, so the maximum number of channels we can assign to the tree is k.

TABLE I
SUMMARY OF PARALLELIZATION TECHNIQUES

		spatial		temporal		utilized multichannel	support		load assignment	remarks(*)
		enc.	dec.	enc.	dec.	for sequential dec.	BSC	MSC		
source file partitioning	block	-	-	yes	yes	no	yes	yes	postprocess	-
	interleave	-	-	yes	yes*	need sync	yes	yes	conflict w/ seq. dec.	may out-of-sync
compressed file partitioning	block	-	-	-	yes*	no	yes	no	strong	high memory requirement
	interleave	-	yes*	-	-	yes	yes	yes	need sync	parallel dec. need many processors
multichannel decoding tree	general	yes	yes*	-	-	yes	no	yes	strong	parallel dec. need many processors
	monotone	yes*	yes*	-	-	yes	no	yes	not the strongest	more spatial-parallel options

video streaming. In other words, MSC is suitable for real-time streaming applications. This is also the reason for using tree-decodable MSC due to its instantaneous decoding ability. An example will be given in Section III-A.

III. SPATIAL AND TEMPORAL PARALLELIZATION

To further investigate the possibilities of MSC, we discuss parallel encoding and decoding methods. In the sense of streaming, the latter part of a file corresponds to the data of a later time. If a source symbol or a codeword only appears in one of the parallelized parts, then the parallelization divides the file in the sense of time and thus it is called *temporal parallelization*. On the other hand, if a source symbol or a codeword appears in multiple parallelized parts at the same time, then it is parallelized across the "spaces", e.g., channels, so it is called *spatial parallelization*.

In this section, we discuss the advantages and disadvantages of various parallelization approaches. Some are simple or existing approaches for BSC which we apply them to multiple channels directly or with slight modifications, while some of them are MSC-exclusive which have not been discussed before. A summary of the whole discussion is given in Table I. On the other hand, MSC supports simple parallel decoding by sacrificing some of the load assignment ability. Note that spatial and temporal approaches can be applied together to utilize more channels and processors.

A. Source File Partitioning

We are considering fixed-to-variable length compression in this paper so we discuss source symbol-aligned partition methods here. A trivial approach is to divide the source file into multiple consecutive blocks, say, the first and the second halves of the source file, so that each block can be encoded and decoded separately and be sent through different channels. This approach allows temporal-parallel encoding and decoding by assigning the blocks to different processors. In other words, the drawback is that it cannot utilize all the channels for sequential decoding as the other channels are sending data of other blocks. Note that if we partition the source file into our desired ratios according to the file size, the compressed data of the blocks likely do not follow these ratios. Instead, this can be handled by partitioning the overall compressed data by these ratios (codeword-aligned), which as a result we have different block sizes. An overhead for synchronizing the block boundaries is required. This approach works for both BSC and MSC.

Another approach is to partition the source file in certain interleaving pattern. For example, every $2i$-th symbol is assigned to the first part and every $(2i+1)$-th symbol is assigned to the second part. This approach also allows temporal-parallel encoding and decoding in a similar manner and it works for both BSC and MSC.

At a first glance, decoding the parts in parallel would obtain source symbols which are close to each other. However, as the codeword lengths are varying, the compressed data among different parts can be out-of-sync. Therefore, it is possible that one part is decoded faster than the other parts. This induces a burden on the memory for buffering the decoded symbols and the decoding may be not sequential. We can resolve this by dynamically assigning the parts in exchange of extra overhead for synchronization. However, we cannot control the load assignment this case unless the source file has some special structure to capture, e.g., the one in the following example. This example also demonstrates that MSC has a faster reception rate for each source symbol as well.

Example 2. Suppose $\mathcal{Z} = \{A, B, C, D, E\}$ and a source file which repeats the string $ABACADAE$ k times for some k. We have $\Pr(A) = 50\%$, $\Pr(B) = \Pr(C) = \Pr(D) = \Pr(E) = 12.5\%$. Consider a 2-channel binary system. The Huffman tree is illustrated in Fig. 2b. Suppose we want to balance the load of both channels. This can be done by assigning every 2 source symbols to the channels alternatively. The compressed files are $(01000110)^k$ and $(01010111)^k$ for channels 0 and 1 respectively. We need to receive 4 bits from channel 0, i.e., 4 units of time, before we can decode every 2 source symbols encoded in this channel. For MSC, the compressed files are $(01010101)^k$ and $(00011011)^k$ for channels 0 and 1 respectively. We only need to receive 2 bits from each of the channels, i.e., 2 units of time, to decode every 2 symbols.

B. Compressed File Partitioning

We now consider the partitioning of the compressed files. The first approach is to partition the compressed files into consecutive blocks so that each block can be sent through different channels. Note that the codeword lengths are not fixed so the codeword boundaries are not explicit, and we have no hint to random access a codeword. Partitioning the compressed file randomly does not guarantee that the cuts are aligned to the boundaries of the codewords.

Klein and Wiseman [9] proposed a heuristic for parallel decoding single-channel Huffman codes, which in general the

idea works for any BSC with decoding trees. In this approach, we first partition the compressed file into blocks without considering codeword alignment. Each block is assigned to a different processor. Each processor decodes its block as if it is an ordinary sequence of codewords, and continues to decode the next block until the boundary of the most recent decoded codeword aligns with that produced by the next processor, where this alignment is called *self-synchronization*. It is possible that we cannot find any aligned codeword in the whole block. In this case, the processor continues to "overflow" the decoding procedure to the next block. However, there are examples which are not self-synchronizing at all. Practically, the destination has to store individual decoded parts before merging as there is no hint on where the decoded parts locate in the source file. Also, some computational resource is wasted on "overflow" decoding and copying during merging. We consider this parallel decoding temporal as its core idea is to divide a file temporally. Load assignment is simple as we can partition the compressed file without considering codeword alignment.

When we apply a similar technique to MSC, each "block" is a tuple where each component corresponds to a part of a distinct compressed file. However, there is no hint to know whether the channel symbols in each component refer to the same codeword. A false self-synchronization, i.e., some components are aligned to different codewords, leads to wrong decoded symbols and cannot be detected immediately. So, we consider this idea cannot be applied to MSC.

Another approach is to partition the compressed file in certain interleaving pattern. At the same time, we can assign the load freely but we need to synchronize the interleaving pattern with the decoder. It is likely that each part is formed by concatenating fragments of codewords. In order to decode, we need to read across parts to reassemble some complete codewords. As we are obtaining sequential codewords after deinterleaving, the heuristic in [9] cannot be applied here. However, we can apply the parallel approach proposed by Johnston and McCreath [10] for BSC. In a nutshell, their algorithm attempts to decode a single codeword starting from each position of the sequence of to-be-decoded channel symbols. A linear scan is then performed to cherry-pick those codewords which are consistent with the previously chosen ones, and discard the others. However, many processors are tasked with useless work which will be discarded later. Also, the computation is much more complex than simply looking up a decoding tree. We consider this parallel decoding spatial as the codewords can be (likely) separated into multiple channels. For example, we can interleave the compressed file in a bit-by-bit manner so that the codewords are separated to multiple channels nicely. This approach can be applied for MSC, but we need to try each position in each channel, where the number of combinations grows quickly.

C. Multichannel Decoding Trees

For any MSC, we can spatial-parallelly encode channel-wisely as long as we have a mapping to map any source symbol to its codeword in any channel. Each channel requires a processor. Algorithm 1 shows this encoding approach. Note that the processors do not need to communicate with each other, so the parallelization can be processed independently. For MSC with a decoding tree with classes assigned, we can create such a mapping by traversing the tree. Algorithm 2 describes the creation of this mapping.

Algorithm 1: Spatial-Parallel Encoding Channel-wisely

```
Data: The channel number i
repeat
    s ← the next source symbol;
    t ← the string of encoded s in channel i;
    Append t to the i-th channel;
until all source symbols are encoded in channel i;
```

Algorithm 2: Creation of Channel-wise Mapping

```
w ← an n-tuple of empty strings;
c ← root node of the tree;
traverse(w, c);
Function traverse(w, c):
    if c is a leaf node then
        s ← the source symbol associated with c;
        Store the mapping that maps s to w;
    else
        j ← the class of c;
        foreach branch i of c do
            w′ ← w;
            c′ ← the node under branch i;
            s ← the channel symbol of branch i;
            Append s to the j-th component of w′;
            traverse(w′, c′);
    return;
```

Before we continue, we define a special type of decoding tree which will be used in the following discussion.

Definition 2. A *monotone tree* is a decoding tree where the class of any internal node is no larger than that of any of its ancestors.

Note that the tree after load assignment may not be a monotone tree. For a non-monotone tree, unless we successfully reach a leaf, we cannot ensure that the codeword does not have any more channel symbols in any of the channels. This means that it is not easy to parallelize decoding, unless we adopt a similar technique as the one proposed by Johnston and McCreath [10] (see the previous subsection), which needs to waste the work of many processors. Nevertheless, this is a possible spatial-parallel decoding approach.

Now, we consider monotone decoding trees. First, it is not surprised that the load assignment ability is weaker than a general tree due to the extra constraint. We continue Example 1 and reassign the classes of the nodes to make the tree monotone. Without loss of generality, let the root node be a class 0 node. If the node above the leaves A and B is of class 0, then the expected codeword length of channel 0 exceeds 1.25. So, this node must be of class 1. Similarly, the sibling of this node must be of class 1. Due to the monotone constraint, the

Algorithm 3: Spatial-Parallel Decoding Channel-wisely

```
Data: The channel number i
repeat
    c ← the first node in Q_i;
    Dequeue c from Q_i;
    c ← decode(c,i);
    if c ≠ next_channel then
        Enqueue c to Q_{i+1}
until the source file is recovered;
Function decode(c, i):
    if c is not a leaf node then
        j ← the class of c;
        while c is not a leaf node and the class of c = j do
            Fetch a channel symbol s from the j-th channel;
            Remove s from the buffer;
            c ← the node under branch s;
    if c is a leaf node and i = n − 1 then
        Append the decoded source symbol to the output file;
        return next_channel;
    return c;
```

Algorithm 4: Spatial-Parallel Decoding Codeword-wisely

```
repeat
    d ← the index of the next unassigned source symbol;
    c ← the root node of the tree;
    for i = 0, 1, 2, ..., n − 1 do
        Wait until M_i = d;
        c ← decode(c,i);
        if c ≠ next_channel then
            M_i ← d + 1
until all source symbols are assigned;
```

Algorithm 5: Spatial-Parallel Encoding Source Symbol-wisely

```
repeat
    d ← the index of the next unassigned source symbol;
    for i = 0, 1, 2, ..., n − 1 do
        Wait until M_i = d;
        s ← the d-th source symbol;
        t ← the string of encoded s in channel i;
        Append t to the i-th channel;
        M_i ← d + 1;
until all source symbols are assigned;
```

remaining nodes in the tree must be of class 1, but then the expected codeword length of channel 1 exceeds 1.25. That is, we cannot balance the load on a monotone tree in this example.

However, the monotone property enables some simple spatial-parallel decoding approaches which have much lower complexity than the Johnston and McCreath-like approach. Specifically, we can parallelize the decoding process channel-wisely or codeword-wisely. On the other hand, besides parallel encoding channel-wisely, we can also do it codeword-wisely. The codeword-wise approaches can use more processors but with a higher synchronization cost. We defer the detail descriptions of these algorithms to the next section.

IV. Parallelization for Monotone Trees

Without loss of generality, assume the root note is of class 0. The first way to spatial-parallelly decode MSC with monotone decoding tree is to spread the computation channel-wisely. As same as Algorithm 1, the number of processors required equals the number of channels. Unlike Algorithm 1, the processors have to communicate with each other as we need to confirm the boundaries of the codewords in each channel. By the monotonicity of the tree, we know that all the channel symbols of the codeword in the current channel are read when we reach a node of another class in the tree. Algorithm 3 shows this decoding approach. To realize it, we need a queue for each channel for synchronization purpose. Denote by Q_i the queue for channel i, where each queue stores (the reference of) a node in the tree. At the beginning, we enqueue the root node k times to Q_0, where k is the number of source symbols in the source file. Equivalently, we can consider Q_0 as a virtual queue that would not deplete the only member (the root node) in it until all codeword symbols in channel 0 are read.

The second way is to spread the computation codeword-wisely as shown in Algorithm 4. The function decode is defined in Algorithm 3. Similar as it channel-wise counterpart, we need to synchronize whether the channel symbols of a codeword are all read from a channel. Therefore, we define an atomic variable M_i for each channel i, indicating which source symbol is currently decoding. At the beginning, all M_i are set to 0. The number of processors are not limited in this approach and they work in a pipelining manner.

Besides the channel-wisely spatial-parallel encoding approach in Algorithm 1, the monotone property enables a source symbol-wise approach as shown in Algorithm 5. We need to ensure that we write the codewords in sequential order, so we define an atomic variable M_i for each channel i for indicating which source symbol is currently encoding. All M_i are set to 0 at the beginning. The number of processors are not limited and they work in a pipelining manner.

Algorithm 6: Heuristic for Load Assignment

```
Data: Desired load ratios (r_i)_{i=0}^{n−1} in decreasing order
h ← height of the tree;
s ← array of h elements;
Calculate the reaching prob. of each node in the tree;
t ← sum of all reaching prob.;
foreach layer ℓ do
    s(ℓ) ← (sum of reaching prob. of nodes in layer ℓ)/t;
ℓ ← 0;
for i = 0, 1, ..., n − 1 do
    r ← Σ_{k=0}^{i} r_k;
    a ← the maximum j such that Σ_{k=0}^{j} s(k) ≤ r;
    b ← the minimum j such that Σ_{k=0}^{j} s(k) ≥ r;
    if a = b or r − Σ_{k=0}^{a} s(k) < Σ_{k=0}^{b} s(k) − r then
        Assign class i to all nodes from layer ℓ to layer a;
        ℓ ← a + 1;
    else
        Assign class i to all nodes from layer ℓ to layer b;
        ℓ ← b + 1;
```

TABLE II
RUNNING TIME EXPERIMENTS (6 PROCESSORS, 6 CHANNELS)

	source file size (bytes)	BSC Huffman encoding	BSC Huffman decoding	channel-wise MSC encoding	channel-wise MSC decoding
Flickr	202,856,867	1540.41	4740.56	1555.19	3604.80
Twitter	1,084,263,840	9413.39	33605.04	8440.08	17417.68
Bible	4,404,443	33.82	125.20	30.76	92.13

We leave the method to efficiently construct a monotone tree which has close-to-desired-load property as an open problem. However, we need to have a class-assigning approach for monotone tree before we can run any experiment. So, we propose a heuristic Algorithm 6 for this purpose. In the algorithm, the height of the tree is defined to be the number of nodes in the path from the root node to the deepest internal node. Without loss of generality, assume the desired load ratio $(r_i)_{i=0}^{n-1}$ in the input is in decreasing order.

The idea of this heuristic is that we first cut the decoding tree layer by layer, where each layer consists of all non-leaf nodes of the same depth. Then, we assign the same class to all the nodes in the same layer. The expected codeword length of the codeword in channel i is the sum of the reaching probabilities of class i nodes in the path from the root node to the leaf node of the codeword. Now, we list the sequence of the sum of reaching probabilities of the nodes in each layer. This sequence is a monotonic decreasing. So, we should assign the top layers to the channel which has the highest load, so on and so forth. Our goal is to partition the sequence into consecutive subsequences such that the sum from the start of the sequence to the end of each subsequence is close to the cumulative ratio we desired.

V. EXPERIMENTS

We consider channel-wise encoding and decoding of Huffman code as an experiment, with a *balanced* desired load among the channels. That is, the code is an *optimal* MSC which is built *efficiently* by:

1) Building a single channel Huffman tree;
2) Assign the classes to the Huffman tree by Algorithm 6 with a balanced desired load.

We use the following corpus in the experiment:

a) The personal tags of Flickr personal taxonomies [14];
b) The Twitter Cheng-Caverlee-Lee scrape [15];
c) The Bible (King James version).

After downloading each corpus, we decompress it and then recompress it as a ZIP file in storage mode, i.e., our experiment considers the compression and decompression of a single file.

For a fair comparison, we precompute the decoding trees and the mappings for encoding. Although the precomputation includes the execution of Algorithm 6, the algorithm is just a linear scan on the decoding tree where the running time is negligible. Also, we load the source file into memory and allocate the pages for outputting the compressed and decompressed data in memory beforehand, so that the running time is not affected by disk I/O nor the kernel buffer cache.

Table II shows the average time (in ms) of 10 runs using 6 processors (6 channels) of an Intel Core i9-9900KS machine with 64GB RAM. We can see that the encoding time of MSC and that of BSC are almost the same, as each processor has to read the whole source file. Unlike the Twitter corpus, there is basically no capital letters in the Flickr corpus. That is, the Twitter corpus has a richer decoding tree, which may be a cause of the improvement for encoding. For the decoding time, we can see that the gain is not in ratio of the number of processors. This is because the processors are working on neighbouring data so they may not be fully utilized due to synchronization.

VI. CONCLUSION

We discussed the applications of multichannel source coding (MSC), compared various parallelization approaches, and proposed a monotone variant of MSC. As a recall, the main advantages of MSC are that it allows a natural way to assign load ratios to the channels, and it can utilize multiple channels in a way that we can achieve a faster reception rate for each source symbol. This makes MSC a potential candidate for multipath or multisource streaming applications.

REFERENCES

[1] H. Yao and R. W. Yeung, "Zero-error multichannel source coding," in *Proc. ITW '10*, Jan. 2010, pp. 1–5.
[2] H. H. F. Yin, K. H. Ng, Y. T. Shing, R. W. F. Lai, and X. Wang, "Polynomial-time construction of two-channel prefix-free codes with given codeword lengths," in *Proc. ITW '21*, Oct. 2021.
[3] ——, "Decision procedure for the existence of two-channel prefix-free codes," in *Proc. ISIT '19*, Jul. 2019, pp. 1522–1526.
[4] H. H. F. Yin, X. Wang, K. H. Ng, R. W. F. Lai, L. K. L. Ng, and J. P. K. Ma, "On multi-channel Huffman codes for asymmetric-alphabet channels," in *Proc. ISIT '21*, Jul. 2021, pp. 2024–2029.
[5] H. H. F. Yin, H. W. H. Wong, M. Tahernia, and R. W. F. Lai, "Multichannel optimal tree-decodable codes are not always optimal prefix codes," in *Proc. ISIT '22*, Jun. 2022, pp. 43–48.
[6] H. Yamamoto, M. Tsuchihashi, and J. Honda, "Almost instantaneous fixed-to-variable length codes," *IEEE Trans. Inf. Theory*, vol. 61, no. 12, pp. 6432–6443, Oct. 2015.
[7] P. G. Howard and J. S. Vitter, "Parallel lossless image compression using Huffman and arithmetic coding," in *Proc. DCC '92*, Mar. 1992, pp. 299–308.
[8] S. T. Klein and Y. Wiseman, "Parallel Lempel Ziv coding," *Discrete Applied Mathematics*, vol. 146, no. 2, pp. 180–191, Mar. 2005.
[9] ——, "Parallel Huffman decoding with applications to JPEG files," *The Computer Journal*, vol. 46, no. 5, pp. 487–497, Jan. 2003.
[10] B. Johnston and E. C. McCreath, "Parallel Huffman decoding: Presenting a fast and scalable algorithm for increasingly multicore devices," in *Proc. ISPA/IUCC '17*, Dec. 2017, pp. 949–958.
[11] E. Ordentlich, M. J. Weinberger, and C. Chang, "On multi-directional context sets," *IEEE Trans. Inf. Theory*, vol. 57, no. 10, pp. 6827–6836, Oct. 2011.
[12] F. Fernández, A. Viola, and M. J. Weinberger, "Efficient algorithms for constructing optimal bi-directional context sets," in *Proc. DCC '10*, Mar. 2010, pp. 179–188.
[13] R. Krichevsky and V. Trofimov, "The performance of universal encoding," *IEEE Trans. Inf. Theory*, vol. 27, no. 2, pp. 199–207, Mar. 1981.
[14] A. Plangprasopchok, K. Lerman, and L. Getoor, "Growing a tree in the forest: Constructing folksonomies by integrating structured metadata," in *Proc. KDD '10*, Jul. 2010, pp. 949–958.
[15] Z. Cheng, J. Caverlee, and K. Lee, "You are where you tweet: A content-based approach to geo-locating Twitter users," in *Proc. CIKM '10*, Oct. 2010, pp. 759–768.