Parallel implementation of a multialphabet arithmetic coding algorithm

*S. Mahapatra, J. L. Núñez, C. Feregrino-Uribe and S. Jones

Electronic Systems Design Laboratory

Department of Electronic and Electrical Engineering

Loughborough University

Loughborough, Leicestershire, LE11 3TU, U.K.

1 Introduction

The need for data compression is growing day by day because of the fast development of data intensive applications that place a heavy demand on information storage and to meet the high data rate requirements of bandwidth hungry transmission systems such as multimedia. Compression techniques remove the redundancy present in the data, keeping only sufficient information that can be effectively utilised in the decompression phase to retrieve the original data. Data compression techniques can be broadly categorised into two major families: (i) lossy and (ii) lossless. Lossy data compression is slightly inaccurate, but in return results in greatly increased compression and is effective while compressing graphics image files or digitised voice which possess enough redundancy. On the contrary, lossless data compression techniques are guaranteed to generate an exact replica of the input file after the decompression phase and find application when storing data base records, spreadsheets, computer programs etc., where lossy compression techniques can not be applied indiscriminately.

The process of lossless data compression can be divided into two distinct classes: (i) dictionary based and (ii) statistical. In dictionary based techniques, groups of consecutive symbols are replaced by a corresponding code. Statistical methods, on the other hand, replace each symbol with a code based on its probability of occurrence. It is convenient to further subdivide statistical methods of data compression into two separate tasks, namely modelling and coding. In the modelling phase, probabilities are estimated for each symbol in the alphabet either statically or in an adaptive manner. In the coding phase, these probabilities are translated into suitable codeword that represent the input symbols. Over the years arithmetic coding [1, 2], has emerged as a versatile and efficient technique for lossless data compression. It scores over the well-established Huffman coding [3] in that it does not need an integral number of bits to encode a symbol.

Arithmetic coding attempts to represent the source data with the minimal entropy. Instead of replacing each bit with its corresponding code as done in traditional techniques, it represents a stream of input symbols by a number in the half-open interval [0,1). At the beginning, the coding interval is set to [0-1). As the encoding progresses, the interval needed to represent the input data stream becomes smaller, needing more bits to represent it. The symbols having a higher probability reduce the range by a smaller amount than a less probable symbol thereby adding fewer bits to the output. As each subinterval attained by encoding a symbol is located within the previous interval, it represents all the symbols encoded so far. Encoding continues recursively until all the input symbols are exhausted. Thereafter, a pointer to the final interval is generated as the final codeword that represents the entire message. In the decoding phase, a similar process is followed to retrieve the original message. It is also possible to generate the code incrementally at the encoder and follow an incremental decoding thereby permitting the operations to proceed on the fly, a feature significant in online data transmission.

As discussed above, arithmetic coding works by reducing the size of an interval proportionately to successive symbol probabilities. This involves the expensive multiplication operation, which is rather slow in both hardware and software. The attempts at replacing the multiplication operation by shifts

* On leave from Regional Engineering College, Rourkela, Orissa, India.

and adds [4]-[5] may lead to a loss in compression efficiency. Attempts have also been made to parallelize the arithmetic coding process thereby resulting in faster implementation. As it has been mentioned in [7], it is possible to parallelize the encoding operation, but not the decoding. The decoding operation involves much more complex data dependencies and generally has to be performed sequentially. Although parallel decoding to a limited extent is demonstrated in [8], it is valid only for binary alphabets.

In the current work, we have divided the multialphabet arithmetic coding operation into a number of stages, each stage needing the encoding of a binary decision. The multiple stages can be executed in a parallel pipelined manner to speedup the encoding operation. Following this scheme, it is possible to parallelize both the encoding and decoding operations that turn out to be identical to each other.

Section 2, following the introductory section briefly discusses the arithmetic coding operation. The structure of the modelling unit is outlined in Section 3, which also presents the overall implementation scheme. Finally, Section 4 concludes this paper with some simulation results.

2 Arithmetic coding

In the following explanation we will directly use integer arithmetic, thereby gaining an insight into the practical implementation of the arithmetic coding algorithm. We start by initialising the coding interval to [0,N), where N depends on the amount of precision we desire. Afterwards, the following steps are to be repeatedly executed to encode the input string.

1. Input a symbol.
2. Subdivide the current interval into a number of subintervals, each corresponding to a different symbol in the alphabet.
3. Select the subinterval corresponding to the symbol input as the new current interval.
4. Execute the normalisation procedure.

The normalisation procedure tests to see if the current interval lies entirely within any of the three intervals: [0,N/2), [N/4,3N/4), or [N/2,N) and acts accordingly as outlined below:

- If the current interval is completely within [0,N/2), 0 is output followed by any 1s left over from earlier events and the current interval is doubled by linearly expanding the interval [0.N/2) to [0,1).
- If the current interval is completely within [N/2,1), 1 is output followed by any 0s left over from earlier events and the current interval is doubled by linearly expanding the interval [N/2,1) to [0,1).
- If the current interval is completely within [N/4,3N/4), this fact is recorded by incrementing a follow count and the current interval is doubled by linearly expanding the interval [N/4,3N/4) to [0,1).

The decoder in a sense recursively undoes the encoder's recursion. It verifies the input bits to determine the interval pointed to by these and finds the symbol, which, when encoded, reduces the coding interval to this interval.

3 The proposed implementation

3.1 Modelling

From discussions in the previous section it is clear that the modeller is required to keep track of the cumulative frequency information and supply to the coder the counts corresponding to the symbol to be coded. A major cost of data compression using adaptive arithmetic coding is to maintain and update these cumulative frequency counts. In [1], Witten et. al. have used different arrays for keeping the counts, recording the cumulative frequency counts and for reordering the counts. The reordering is required to ensure that the frequency counts are in decreasing order so that the symbol being processed is more likely to be towards the left and would require relatively fewer increments of the cumulative frequency counts. This implementation works well for files which have a skewed character usage. But, for others like object files where the character usage is less skewed, the method performs poorly. Also, it is suitable for a relatively smaller alphabet, the running time becoming prohibitive for large alphabets. To overcome these shortcomings Moffat [6] has proposed an alternative implementation where the symbols are visualised as the nodes of a binary tree of $\lfloor \log_2 N \rfloor + 1$ levels, where N is the alphabet size. Each node is associated with a particular symbol and records its position in the tree along with the current code space allocated to that symbol. Using this structure the cumulative counts can be obtained for any symbol in order $O(\log_2 N)$ time. Two pointers are used in this implementation to keep the nodes arranged in decreasing order of occurrence counts. Although this algorithm has a linear complexity in terms of the output code length, it may prove expensive for hardware realisation due to the excessive storage space it requires.

In the proposed implementation, we use a slightly different approach to store, retrieve, and update the model related information. The data structure used is still a complete binary tree, but now symbols are assigned only to the leaf nodes. Let C(s) denotes the cumulative frequency count for symbol 's'. The model related information is now implicitly stored in a single variable at each node. The information stored in node j at level i of the tree (node N(i,j)) is given by:

$$D(i,j)=C[(2j+1)2^{n-i-1}]-C[(2j)2^{n-i-1}],\ 0\le i\le n-1,\ 0\le j\le 2^{i}-1. \qquad (1)$$

One such binary tree for 16 symbols is depicted in Figure 1. In addition to the information stored in each of the nodes, the model requires one more information, i.e., the total count T that is stored in a separate register.

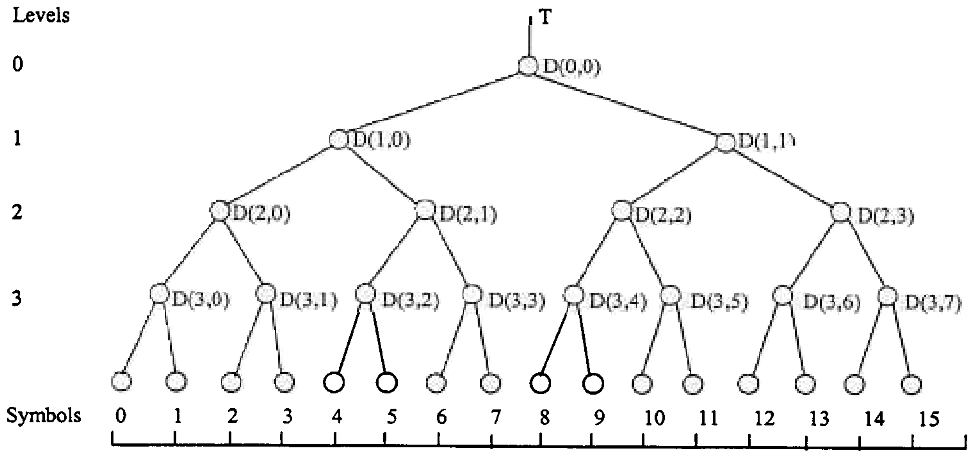

Figure 1 Coding tree for 16 symbols

Adaptive modification of the model related information is easy to perform. When any symbol is encountered, the information stored in a node N(i,j) along the path from the root to the symbol is incremented only if the ith bit b_i is zero. This achieves the necessary adaptation in the code space.

3.2 Coding

The manner in which the cumulative frequency information is stored has a direct bearing on the proposed encoding algorithm. The principal idea behind the algorithm is that here the multialphabet arithmetic coding is performed with the help of multiple binary coders, each at a different level of a binary tree. We move downwards from the root of the tree and the coding information supplied at each level depends on the information stored at the nodes in the path leading to the symbol. The encoding algorithm is as follows:

Algorithm 1

```
1. To start with, initialise the cumulative frequency C(i)=i and the other nodes accordingly.
2. The total count T enters the root (node N(0,0)) as the top value T(0,0).
3. For i=0 to n-1, the following steps are repeatedly executed in a node N(i,j) along the path.
    i.  If $b_i$ =0, send the value D(i,j) to the left child, i.e., to node N(i+1,2j) as the top value.
    ii. Else, send the value T(i,j)-D(i,j) to the right child, i.e., to node N(i+1,2j+1) as the top
        value.
```

iii. Send the values (D(i,j),T(i,j)) as the coding information to the coder corresponding to that level. Update D(i,j)=D(i,j)+(1-b_i).

3.3 Implementation

As far as the modelling unit is concerned, it can be implemented as a linear array of systolic units (Figure 2). Each unit stores all the nodes at the corresponding level of the tree, i.e., M_0 stores the root, M_1 stores the two nodes in level 1 of the tree and so on. For encoding an alphabet involving 256 symbols, the modeller will have a total of 8 different units that are linearly interconnected as shown. It is clear that the memory requirements of the nodes will be skewed in nature. Each unit in the modeller delivers the coding information to an arithmetic coder. So, there are a total of 8 coders (Q_0 to Q_7), each encoding the information given out by the corresponding modelling unit.

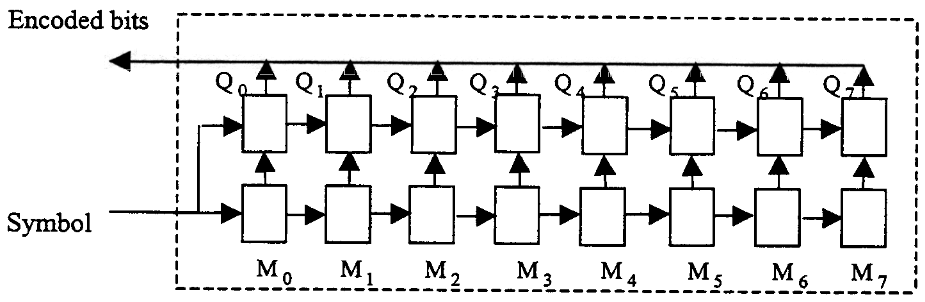

Figure-2 Parallel arithmetic encoder

The encoding of the input symbols proceeds as outlined in Algorithm 2.

Algorithm 2

1. The symbols enter the encoding unit from the left. As it is observed the symbols go through both the modelling units and the coding units.
2. As a symbol s moves from M_0 to M_7, each unit M_i performs the following operations:
   - i. It reads the bit b_i in s to determine what operation to perform (Algorithm 1).
   - ii. It then transfers the necessary coding information to the corresponding coder.
   - iii. It updates the differential cumulative frequency count corresponding to the first *i*-1 bits in the symbol, depending on b_i.
   - iv. It transfers the symbol to the next modelling unit.
3. Each of the coder Q_i performs the following operations:
   - i. It receives the coding information from the corresponding modeller and encodes either the left or the right half of the coding interval according to the bit b_i of the symbol. The coding information received gives the middle and right extreme of the coding interval. It uses 0 as the left extreme.
   - ii. The resulting code bits if any are shifted into an 8 bit temporary buffer. When this 8 bit buffer becomes full, its content is transferred to a second 8 bit output buffer. When the output buffer of each of the coding units is full, these are sent out together as the coded information.

The above algorithm can be executed in a pipelined manner to encode all the symbols in the input data stream.

At the decoder side we have the same array structure for both the decoding and the modelling units (Figure 3). The encoded bits are segregated into 8 bit bytes and given to corresponding decoding units. Each unit decodes a sequence of left and right decisions or 0 and 1 bits. Combination of the corresponding decisions at all the units gives the decoded symbol. It is clear that decoding can also be performed in a parallel pipelined manner.

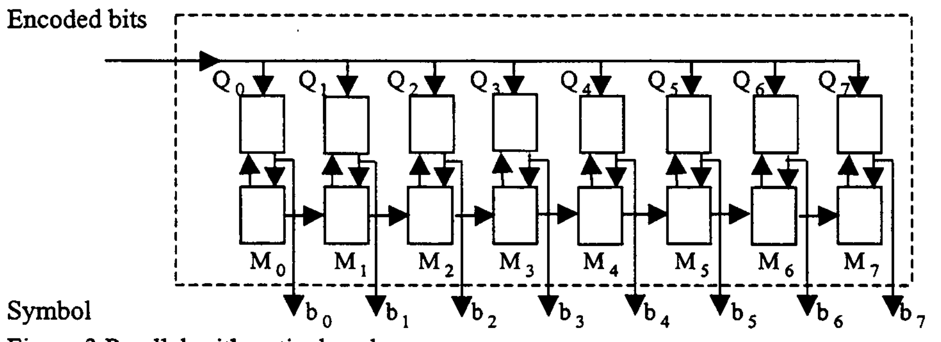

Figure 3 Parallel arithmetic decoder

4. Conclusion

This paper presents the parallel pipelined implementation of a multialphabet arithmetic coding algorithm. Unlike the existing implementations, here, the encoding of any symbol is performed at multiple levels of a binary tree representing the symbol and thus it is possible to parallelize both the encoding and decoding operations with multiple coders or decoders working at different levels of the tree. The proposed scheme also eases the hardware realisation of the entire module.

Table 1 enlists the simulation results for different types of files in the Canterbury Corpus benchmark set of programs. In this, a comparison is made between the compression efficiencies obtained while compressing these files using a sequential arithmetic coder and the proposed arithmetic coder in 4kB blocks. Though a slight degradation is observed in the compression efficiency, this may be attributed to the use of approximations in arithmetic coding and can be further reduced by increasing the precision of the arithmetic operations.

Table 1. Simulation results.

File	Size, bytes	Compression Efficiency		
		Seq. Coder	Proposed Coder	% Degradation
Text	152,089	0.6132	0.6186	0.88
HTML	25248	0.7003	0.7049	0.65
Play	129,301	0.6494	0.6542	0.74
Sparc Exec.	38,240	0.6105	0.6128	0.38
Src. Prog.	11,581	0.6632	0.6685	0.80

References

1. I. H. Witten, R. M. Neal and J. G. Cleary, Arithmetic coding for data compression, Commn. ACM, vol. 30, no. 6, June 1987, pp. 520-540.
2. P. G. Howard and J. S. Vitter, Arithmetic coding for data compression, Proc. of IEEE, vol 82, no. 6, Jun. 1994, pp. 857-865.
3. D. A. Huffman, A method for the construction of minimum redundancy codes, Proceedings of the Institute of Radio Engineers, vol. 40, 1952, pp. 1098-1101.
4. J. Rissanen and K. M. Mohiuddin, A multiplication free multialphabet arithmetic code, IEEE Trans. Commun., vol 37, no. 2, Fb. 1989, pp. 93-98.
5. S. M. Lei, Efficient multiplication-free arithmetic codes, IEEE Trans. Commun., vol. 43, no. 12, Dec. 1995, pp. 2950-2958.
6. A. Moffat, Linear time adaptive arithmetic coding, IEEE Trans. Inform. Theory, vol. 36, no. 2, Mar. 1990, pp. 401-406.
7. J. Jiang and S. Jones, Parallel design of arithmetic coding, IEE Proc. – Comp. And Dig. Tech., Sep. 1994, vol. 144, no. 6, pp. 327-333.
8. J. Jiang, A novel parallel architecture for black and white image compression, Signal Processing Image Commun. 8 (1996), pp. 465-474.
9. M. H. Hsieh and C. H. Wei, An adaptive multialphabet arithmetic coding for video compression, IEEE Trans. On Circuits and Sys. for Video Tech., vol. 8, no. 2, Apr. 1998, pp. 130-137.