Parallel Lossless Image Compression Using Huffman and Arithmetic Coding

Paul G. Howard and *Jeffrey Scott Vitter*

A shorter version of this paper appears in *Proceedings of the IEEE Data Compression Conference*, Snowbird, Utah, March 23–26, 1992, 299–308.

Parallel Lossless Image Compression Using Huffman and Arithmetic Coding[1]

Paul G. Howard[2]
Visual Communications Research
AT&T Bell Laboratories
Holmdel, N.J. 07733–3030

Jeffrey Scott Vitter[3]
Department of Computer Science
Duke University
Durham, N.C. 27706–0129

Abstract

We show that high-resolution images can be encoded and decoded efficiently in parallel. We present an algorithm based on the hierarchical MLP method, used either with Huffman coding or with a new variant of arithmetic coding called *quasi-arithmetic coding*. The coding step can be parallelized, even though the codes for different pixels are of different lengths; parallelization of the prediction and error modeling components is straightforward.

Index terms: Data compression, Huffman coding, arithmetic coding, parallel algorithms.

[1]A shorter version of this paper appears in *Proceedings of the IEEE Data Compression Conference*, Snowbird, Utah, March 23–26, 1992, 299–308.

[2]Work was performed while the author was at Brown University and at Duke University. Support was provided in part by NASA Graduate Student Researchers Program grant NGT–50420 and by a National Science Foundation Presidential Young Investigator Award grant with matching funds from IBM. Additional support was provided by a Universities Space Research Association/CESDIS associate membership.

[3]Work was performed in part while the author was at Brown University. Support was provided in part by a National Science Foundation Presidential Young Investigator Award grant with matching funds from IBM and by Air Force Office of Scientific Research grants F49620–92–J–0515 and F49620–94–1–0217. Additional support was provided by a Universities Space Research Association/CESDIS associate membership.

1 Introduction

The increasing availability of massively parallel computing architectures and the enormous volume of scientific data being produced make parallel data compression a natural subject of research. We present general-purpose algorithms for encoding and decoding using both Huffman and arithmetic coding, and apply them to the problem of lossless compression of high-resolution grayscale images.

Our system for lossless image compression has four components [8]: pixel sequence, prediction, error modeling, and coding. In [8] we introduced MLP, a multi-level progressive method for lossless image compression. The pixel sequencer of MLP divides an $m \times m$ image into $2 \log_2 m$ levels, the pixels in each level forming a checkerboard pattern; each level contains twice as many pixels as the preceding one. The prediction of intensity values of pixels at one level depends only on the values of pixels at earlier levels, so it is possible to predict all the pixels at one level in parallel. In this paper we show that the coding step in MLP can also be parallelized.

A distinguishing characteristic of compression algorithms is the need to maintain decodability: the encoder may use only information that will be available to the decoder. Since we are considering the case where we wish to do both encoding and decoding in parallel, the location of bits in the encoded file must be computable by the decoding processors, preferably before they begin decoding.

In the MLP algorithm, the prediction error at each pixel is modeled as a random variate from a particular distribution (Laplace, normal, or some similar distribution) with zero mean; the variance is estimated by the error modeler. We then use a statistical coder (such as Huffman coding or arithmetic coding) to encode the error with respect to the distribution. Since the variance is the only parameter of the distribution and since the code length is not sensitive to small differences in the estimated variance, we can select a small number of variances (about 50) and precompute a distribution (and perhaps a code) for each of them. Thus in the parallel coder, each pixel is encoded using a known fixed code. In Section 2 we develop a parallel encoder and decoder for the simpler Huffman codes; in Section 3 we extend our coders to a practical version of arithmetic coding called *quasi-arithmetic coding*.

We use the PRAM model of parallel computation; the number of processors is p. We allow concurrent reading of the parallel memory, and limited concurrent writing, to a single one-bit register, which we call the *completion register*. At the beginning of each time step it is initialized to **0**; during the time step any number of processors (or none at all) may write **1** to it. Our algorithms are also ideally suited for implementation on the Connection Machine architecture. We code the image by levels; each level consists of n pixels.

The main issue in parallel coding is in dealing with the differences in code lengths of different pixels. For simplicity we do not consider input data routing, and we defer the issues of parallel prediction and error modeling to Section 4, where they are discussed briefly.

2 Parallel Huffman Coding

We now develop the basic parallel coding algorithms using Huffman coding [14]. In practice Huffman coding is often the best choice for statistical coding, since the resulting codes are usually close to optimal and they can be very fast if implemented by lookup tables. Huffman codes are instantaneous; that is, the receiver knows immediately when a complete symbol has been received, and does not have to look further to correctly identify the symbol. Instantaneous codes are just those with the *prefix property*: no code word is a prefix of another code word [7, pages 53–55]. Parallel coding is simplified since the code bits for all pixels are disjoint and independent.

2.1 Codeword-length coding

One simple approach involves assigning one pixel to each processor and noting that each encoding processor can easily compute the code length of its pixel. By a single prefix operation each processor can compute the location of its code bits in the output stream; it can then write them directly. When all processors have finished, a new pixel is assigned to each processor. This method works if

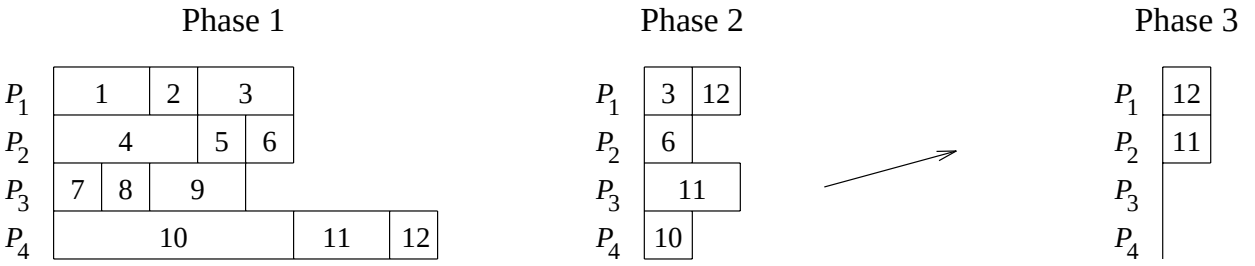


Figure 1: Example of reallocation coding. In this example, there are four processors, P_1, P_2, P_3, P_4 and 12 pixels, $1, 2, 3, \ldots, 12$. Initially three pixels are assigned to each processor. The first phase ends when processor P_3 finishes pixel 9 after 4 time steps; each processor has output 4 bits by this time. Pixels 3 and 10 have been partially encoded; they remain assigned to processors P_1 and P_4; the untouched pixels (6, 11, and 12) are reassigned to balance the remaining load. The second phase ends one time unit later, when P_2 finishes pixel 6 and P_4 finishes pixel 10. The in-progress pixel (11) temporarily remains assigned to P_3; the untouched pixel (12) is "reassigned" to P_1; then since P_2 is inactive, P_3's processing is shifted to P_2. Pixels 11 and 12 are finished in the last time step.

the decoding is to be sequential; in fact it produces the same coded output as a sequential encoder. Parallel decoding is difficult, since the decoding processors cannot easily determine the lengths or the starting locations of the output codewords, although in fact De Agostino and Storer [2] show that this can be done by assigning processors to encoded bits.

2.2 Bit-transpose coding

We can achieve decodability by rearranging (transposing) the output bits. Instead of outputting all bits for the first pixel, then all bits for the second, and so on, we output the first bit for each pixel in the first time step, then the second bit for each pixel in the second time step, and so on. After each time step, we can determine whether any codes are complete by having all completing processors concurrently write **1** to the completion register. If any codes are complete, the corresponding processors no longer produce output; output from the remaining processors is shifted accordingly by a prefix operation. This code can be decoded in parallel: each decoding processor can determine when the code for its pixel is complete, and the same concurrent write and prefix operation can determine where the remaining processors are to obtain their next bits from. When all processors have completed their output, a new pixel is assigned to each processor. If there is one processor for each pixel, this method makes good use of available processors. The difficulty lies in the time-consuming $O(\log p)$ prefix operation that must be performed after every time step in which a processor finishes.

2.3 Reallocation coding

If the number of processors is much less than the number of pixels, we can reduce the number of prefix operations by allowing each processor to begin working on another pixel as soon as it has completed a pixel. At the beginning of a level we distribute the n pixels equally among the p processors. For analysis we assume that the allocation is random, but that the encoder and decoder can make the same allocation without any side information.

During processing, we follow the bit-transpose protocol of Section 2.2: at each time step each processor writes a single bit to a location that will be known to both encoder and decoder. When a processor finishes outputting the code for one pixel, it goes on to the next pixel allocated to it, even though other processors may still be working on earlier pixels. Code lookup is assumed to be fast: the codes are small enough that the full code can be stored in each processor's memory, or perhaps distributed among small groups of nearby processors. When one processor finishes all its allocated pixels, it indicates this fact by writing **1** to the completion register. When the completion

register is **1**, the output process is interrupted. At this time the pixels allocated to other processors fall into three categories: completed, in-progress, and untouched. Completed pixels are of no further concern; in-progress pixels remain assigned to their current processors; untouched pixels are distributed evenly among all the processors. Processing of the reallocated pixels then continues until the next time that some processor finishes all its allocated pixels. Toward the end of processing for a level, the number of remaining pixels will become less than the number of available processors. At this point we deactivate processors, making the number of active processors equal to the number of remaining pixels. We then revert to the bit-transpose method of Section 2.2. The entire process is illustrated in Figure 1.

2.4 Analysis

We analyze the time required by the parallel algorithm. We assume that in one time unit each processor can output one bit, and that a prefix operation, as required for reallocation, takes $2\lceil \log_2 p_t \rceil$ time units, where p_t is the number of active processors at time t. We denote the number of bits in the longest single code word by L, and the average number of bits per pixel in a level by H.

We can easily show that the time required for bit output in one level is between $\lceil nH/p \rceil$ and $\lceil nH/p \rceil + L$. The processors are operating at full efficiency until there are fewer pixels than processors, using at most $\lceil nH/p \rceil$ time units. At that time no processor has more than L bits remaining, so no more than L additional time units are needed. It is clear that even if the processors can operate at full efficiency throughout a level, they require at least $\lceil nH/p \rceil$ time units.

The more interesting analysis concerns the number of reallocations required. We define a *phase* to be the period between reallocations. *Early phases* are those which take place while the number of pixels is greater than the number of processors; *late phases* are those needed to code the final p or fewer pixels. We show that the number of reallocations needed when coding a level with n pixels using p processors is at most $L \log_2(2n/p)$ in the worst case.

We define a *superphase* to be the time needed to halve the number of remaining pixels. Consider the first superphase. The number of pixels assigned to each processor ranges from n/p in the first phase down to $n/2p$ in the last. At least one processor completes all its pixels in each phase; such a processor must output at least one bit per pixel, since all code words in a Huffman code have at least one bit. This processor (and hence all processors) thus output at least $n/2p$ bits in each phase, making a total of at least $n/2$ bits output in each phase. The total number of bits that must be output in the first superphase is at most $nL/2$, so the number of phases in the first superphase is at most L.

The same reasoning holds for all superphases. The number of superphases needed to reduce the number of remaining pixels from n to p is $\log_2(n/p)$, so the number of phases needed is just $L \log_2(n/p)$. Once p or fewer pixels remain, we fall back to bit-transpose coding, which may require L late phases, so the total number of phases needed is at most $L \log_2(2n/p)$.

2.5 Parallel Huffman coding in practice

We have simulated parallel Huffman compression for a set of 14 Landsat Thematic Mapper images; these images, described in [8], are 512×512 8-bit grayscale images. We simulate only the last level of coding ($n = 131{,}072$ pixels) for each image, using $p = 4{,}096$ processors. For our test images, the number of early phases is at most 7, the average being 5.6. It usually happens, however, that the remaining code lengths take on many of the possible values, so the number of late phases is large. The average value of L in the simulations is 23.7, and the average number of late phases is 13.7.

To improve compression time, we must reduce the number of phases. Using the variability index technique described in Section 4.2, we can more evenly balance the output bits among the processors. The result is a reduction in the number of early phases for most of the images; a few stayed the same. The average falls to 4.6. The number of late phases is essentially unaffected.

By reducing L, the length of the longest code, we can reduce the number of late phases. One

straightforward approach is to build optimal code trees subject to a constraint on the length of the longest codeword, as described in [5,13,19]. A simpler *ad hoc* method is to substitute a special IGNORE code for the codes of pixels longer than a certain threshold θ. These *long pixels* must then be transmitted separately. There are not very many of them and their code lengths are long, so we lose very little compression efficiency by sending them unencoded at the end of the level; this is easy to do in parallel after a prefix operation to assign long pixels to processors. The IGNORE code itself can be of length θ. In simulations with $\theta = 10$ using the variability index technique, the average number of late phases falls to 9.1. The average loss in compression is only 196 bytes.

We can further reduce the number of early phases by performing local reallocations. Instead of using time $2\lceil \log_2 p_t \rceil$ to reallocate all untouched pixels when one processor completes its pixels, we can arrange local exchanges between neighboring processors, thus lengthening the time between full reallocations.

2.6 Extension to other prefix codes

Although Huffman codes are optimal among prefix codes for a given distribution, it is sometimes desirable to use simpler prefix codes. Golomb codes [6] and Rice codes [17] can be used to encode data from distributions in which the probabilities are arranged in approximately decreasing order. This condition usually holds for the prediction errors encountered in lossless image compression.

Golomb codes are parameterized by a positive integer parameter m. Given the value of m, we encode non-negative integer n by encoding $\lfloor n/m \rfloor$ in unary, then encoding $n \bmod m$ using an adjusted binary code for the range $[0, m-1]$. It can be shown that the correct choice of the parameter m produces an optimal prefix code for a given exponential distribution [4], but Golomb codes are useful for other decreasing distributions as well.

Rice [17] independently discovered the special case of Golomb codes where $m = 2^k$ for some integer k. Restricting m to be a power of 2 leads to codes that are easy to implement in hardware or software. Given k, the value of the coding parameter, we encode n by first removing the k least significant bits of n and encoding the remaining bits as a unary number. Then we send the k least significant bits directly. Like Golomb codes, Rice codes are most closely matched to exponential distributions but useful for other decreasing distributions.

Because they are prefix codes, Golomb and Rice codes can be implemented in parallel using the reallocation coding method of Section 2.3. The analysis in Section 2.4 applies as well, except that the average number of bits per pixel in a level and the length of the longest codeword may be larger.

3 Parallel Quasi-Arithmetic Coding

Huffman coding produces an average code length close to the entropy of the source model used for coding. In fact, Huffman coding is optimal among prefix codes. The suboptimality of Huffman coding can be appreciable, however, whenever one input event has a probability near 1. In image compression, this happens when the variance of the Laplace distribution describing the model is small, in which case the zero-error event has high probability. For example, when the variance of a Laplace distribution is less than 1.04, the probability of a zero error is more than 0.5; when the variance is less than 0.26, the probability of a zero error is more than 0.75.

When Huffman coding is inadequate, we can turn to arithmetic coding. Arithmetic coding can theoretically achieve exactly optimal compression for a given source model when it is implemented using exact (slow) arithmetic. Practical implementations of arithmetic coding use fixed precision arithmetic [9,11,20], but they still run slowly because of the multiplications (and sometimes divisions) required. Recent research has focused on approximations to the arithmetic that reduce the time required without sacrificing much coding efficiency. Work by Rissanen and Mohiuddin [18], Chevion *et al.* [1], Feygin *et al.* [3], and Printz and Stubley [16] has involved approximate multiplication; Neal [15] uses approximate division. In [9,10,11] we present complete details of an

alternative practical approach, called *quasi-arithmetic coding*, in which we precompute all multiplications and divisions and store the results in lookup tables. We review quasi-arithmetic coding in Section 3.1. Quasi-arithmetic coding can be viewed as a generalization of Huffman coding, so the extension in Section 3.2 of the algorithm in Section 2.3 is natural.

3.1 Quasi-arithmetic coding

In ordinary arithmetic coding we encode a sequence of input events by identifying each possible sequence with a subinterval of the real interval $[0, 1)$, then selecting the subinterval corresponding to the actual input. In practice we use integer subintervals of the interval $[0, N)$ for some sufficiently large N. Whenever we know that the current interval lies entirely in the left or right half of the full interval, we output **0** or **1** respectively, discard the unused half of the full interval, and expand the remaining half so that it fills the full interval. (We also use a trick, the *bits-to-follow* mechanism explained in [20], that enables us to expand the interval when we know that the current interval is entirely within the middle half of the full interval.)

The intermediate intervals computed by the coder can be thought of as states, each determined by the endpoints of the interval. A full precision arithmetic coder has an infinite number of possible states; a practical coder based on interval $[0, N)$ uses $3N^2/16$ states. The idea of quasi-arithmetic coding is that by using a small value of N we can make the number of states small enough that the coder can be represented in lookup tables. The only arithmetic involved is in precomputing the tables; the arithmetic is done according to the Witten-Neal-Cleary algorithm [20].

A one-state coder corresponds to Huffman coding. As we increase the number of states, we increase the precision of the coder and hence its compression efficiency. Using just a few states often provides efficiency considerably greater than that of Huffman coding. Quasi-arithmetic codes, like arithmetic codes, are not instantaneous codes; nevertheless, they are uniquely decodable with bounded coding delay.

Construction of an optimal quasi-arithmetic code for a multi-symbol alphabet is an open problem. However, we can use the following three-step process to construct a reasonable multi-symbol quasi-arithmetic code.

1. We design a multi-state code table for a binary alphabet with various possible symbol probabilities.

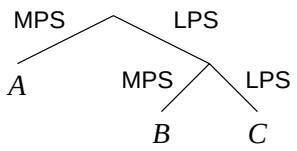
2. We decompose the multi-symbol alphabet into a binary tree with the alphabet symbols at the leaves, such that the product of the edge probabilities from the root to each leaf approximately equals the corresponding symbol probability.

3. We follow all paths through the tree of Step 2, using the table from Step 1 to compute an output codeword and next state for each alphabet symbol from each possible starting state.

Example 1: We construct a two-state code based on full interval $[0, 4)$ for the three-symbol alphabet $\{A, B, C\}$, with $p_A = 0.7$, $p_B = 0.2$, and $p_C = 0.1$. First we construct a two-state binary code. In this example we use a very simple binary code, in which we merely distinguish the input symbols as more or less probable; a more complete two-state code would also allow the two probabilities to be approximately equal. We indicate the more probable symbol by MPS and the less probable symbol by LPS; the probability of MPS can be taken to be about 0.71.

From state	MPS input		LPS input	
	Output	Next state	Output	Next state
E_0	-	E_1	**00**	E_0
E_1	**1**	E_0	**01**	E_0

State E_0 corresponds to the full interval $[0,4)$, state E_1 to subinterval $[1,4)$. Next we construct a binary tree with each branch labeled either MPS or LPS, the probability of MPS being 0.71.



he effective probabilities of the leaves are $p_A^* \approx 0.710$, $p_B^* \approx 0.206$, and $p_c^* \approx 0.084$. Next we use the tree and the table to derive the following code.

From state	Input A		Input B		Input C	
	Output	Next state	Output	Next state	Output	Next state
E_0	–	E_1	**00**	E_1	**0000**	E_0
E_1	**1**	E_0	**01**	E_1	**0100**	E_0

Again, state E_0 corresponds to the full interval $[0,4)$, and state E_1 to subinterval $[1,4)$. Even though it does not have the prefix property, this code is uniquely decodable with bounded delay. For example, from state E_0, the code for B (**00**) is a prefix of the code for C (**0000**); but a B input leads to state E_1, and the first two bits output from state E_1 are never **00**; hence a B will not be decoded as a C. The coding delay is at most one symbol, and the following four-state table can perform the decoding.

From state	Input **0**		Input **1**	
	Output	Next state	Output	Next state
D_0	–	D_1	AA	D_0
D_1	–	D_2	A	D_2
D_2	–	D_3	BA	D_0
D_3	C	D_0	B	D_2

The asymptotic average code length for this code is 1.171 bits per symbol. The entropy of the source is 1.157 bits per symbol, so the compression loss[4] is only $1.2\frac{\circ}{\circ}$. The Huffman code for this source has an average code length of 1.3 bits per symbol, giving a compression loss of $10.5\frac{\circ}{\circ}$, over 8 times as large. In this example we used a two-state code; using more states usually gives even more efficiency. □

Note that the two-symbol code is used only for constructing the multi-symbol code. The multi-symbol code tables and coding algorithm are similar to the Huffman tables and algorithm, but with added state information. In MLP we would precompute the set of Laplace distributions, then precompute a quasi-arithmetic code for each of them.

3.2 Parallel algorithm

We can apply the reallocation coding method of Section 2.3 directly to quasi-arithmetic coding. The only complication arises when the last pixel of a processor's allocation leaves the processor in a state other than the starting state. To ensure decobability, we must not allow this. Naïvely reverting to a Huffman code for the last pixel of a processor's allocation, does not work: toward the end of processing, a pixel with a long code may *become* a processor's last pixel through reallocation, even though it was not the last when the processor began working on it. This happened to pixel 10 in the example of Figure 1.

[4] *Compression loss* can be derived from the definition of compression gain in the revised version of [12] to be $100 \log_e$(actual average code length/entropy); it is expressed in units of *percent log ratio*, denoted by the $\frac{\circ}{\circ}$ symbol. For small losses, the compression loss is almost equal to the inefficiency expressed as an ordinary percentage.

A successful approach is to force each processor back to the starting coder state after its last allocated pixel has been encoded by outputting zero, one, or two additional bits according to the following rule:

> If the current interval is $[0, N)$, do nothing;
> otherwise if $[0, N/2)$ is entirely within the current interval, output **0**;
> otherwise if $[N/2, N)$ is entirely within the current interval, output **1**;
> otherwise if $[N/4, N/2)$ is entirely within the current interval, output **01**;
> otherwise if $[N/2, 3N/4)$ is entirely within the current interval, output **10**.

Example 2: If we were using the three-symbol code in Example 1, and if the last symbol to be output by a processor (from state E_0) were B, we would output **00**; then we would have to force the processor from state E_1 back to state E_0. This can be done by outputting **1** or **01**; according to the third line of the rule above, we output the shorter string **1** since state E_1 (interval $[1, 4)$) includes the entire right half of the full interval $[0, 4)$. Without the extra bit, the decoder would not know whether the last symbol was B or C, since the codes for both of them begin with **00**. After reading **001** and decoding B, the decoder would know (by counting decoded symbols) that there was no more data, so it would not attempt any further decoding; in particular it would not output an extra A. □

In effect, we have a number of arithmetically coded output streams; we have to solve the end-of-file problem for each of them. It is not difficult, merely a nuisance.

The algorithm for parallel quasi-arithmetic coding is as follows:

1. We distribute all the pixels of a level evenly among all the processors. The pixels may be assigned randomly or, even better, we may attempt to give each processor approximately the same number of bits to output, as discussed in Section 4.2.

2. Each processor proceeds sequentially through its assigned pixels, writing one bit to a pre-computed location at each time step. If a pixel completes all its assigned pixels, and it is in the starting state, it writes **1** to the completion register. If a completing processor is not in the starting state, it begins the finishing-up procedure described above. After finishing up, it writes **1** to the completion register (unless a reallocation has taken place, giving the processor more pixels to work on).

3. When the completion register becomes **1**, processing is interrupted for pixel reallocation. Pixels currently being processed remain with their current processor. The untouched pixels are divided among all the processors. Processors that have begun but not completed the finishing-up procedure must complete the procedure.

4. If a reallocation leaves any processors with no pixels and no finishing up to do, those processors are deactivated. We perform a prefix operation on the remaining pixels to determine the location of each processor's next output bit.

After reallocation, we return to Step 2, and repeat until no pixels of the current level remain.

4 Parallel Prediction and Error Modeling

In order to code images using the MLP algorithm, we must predict the value of each pixel and model the error of our prediction. Both of these steps can be parallelized.

4.1 Prediction

The prediction step for a pixel involves computing a linear combination of the values of a fixed constellation of nearby pixels whose values are already known. Clearly the encoder, having access to all pixel intensities at the beginning of the computation, can predict all values simultaneously;

the decoder can make the same predictions, but only level-by-level, since the predictions depend on values from preceding levels.

4.2 Error modeling

Error modeling is most effective when done implicitly. In [12] we give an implicit method for estimating local image variances that leads to better compression than any other published lossless image compression method. Our method involves computing a quantity called the *variability index* for each pixel, sorting the pixels by variability index, then using the same error model (i.e., the Laplace distribution with the same variance) to encode all pixels with similar variability index. Like the intensity prediction, the variability index computation depends only on the values of a few nearby pixels, known from a previous level, so it can be done in parallel by both encoder and decoder with no loss of efficiency. The assignment of variances to values of the variability index can be done implicitly in a sequential environment, adaptively estimating the variance while working through the pixels in order of variability index. For parallel coding the use of side information is more appropriate: we can group the pixels after sorting (sorting can be done efficiently in parallel), then compute the variance of each group and transmit it in coded form. Each variance requires only a few bits to transmit, typically four or fewer; if we divide each level of n pixels into $\sqrt{n}$ groups, only about $(\sqrt{2}+1)\,m$ variances must be transmitted for an $m \times m$ image. For a 512×512 image this amounts to 1236 variances, or only 618 bytes of side information. This introduces less than $0.5\frac{o}{o}$ compression loss.

The variability index technique has the effect of classifying pixels by local variance. This translates roughly into a classification by code length, since distributions with larger variances usually have larger average code length. Using this classification we can assign pixels to processors in a way that divides the *code length* (not just the number of pixels) approximately evenly among the processors, thus reducing the number of pixel reallocations needed. The simulation results described in Section 2.5 confirm the usefulness of this technique.

5 Conclusions

We have shown that efficient parallel lossless encoding and decoding of images is feasible, using either Huffman coding, other prefix codes, or, more surprisingly, a version of arithmetic coding; we have presented algorithms and analysis. Our algorithm uses randomization to limit the number of reallocations, but the variability index technique provides sufficient balancing of output bits to obviate randomization. We note that these algorithms could be implemented on the Connection Machine architecture without difficulty.

Although we have presented our algorithms in terms of image compression, the ideas extend to any compression problem in which the model needed to encode each event is known ahead of time. It is not necessary that each event use the same model.

References

[1] D. Chevion, E. D. Karnin & E. Walach, "High Efficiency, Multiplication Free Approximation of Arithmetic Coding," in *Proc. Data Compression Conference*, J. A. Storer & J. H. Reif, eds., Snowbird, Utah, Apr. 8–11, 1991, 43–52.

[2] S. De Agostino & J. A. Storer, "Parallel Algorithms for Optimal Compression using Dictionaries with the Prefix Property," in *Proc. Data Compression Conference*, J. A. Storer & M. Cohn, eds., Snowbird, Utah, Mar. 24-26, 1992, 52–61.

[3] G. Feygin, P. G. Gulak & P. Chow, "Minimizing Error and VLSI Complexity in the Multiplication Free Approximation of Arithmetic Coding," in *Proc. Data Compression Conference*, J. A. Storer & M. Cohn, eds., Snowbird, Utah, Mar. 30-Apr. 1, 1993, 118–127.

[4] R. G. Gallager & D. C. Van Voorhis, "Optimal Source Codes for Geometrically Distributed Integer Alphabets," *IEEE Trans. Inform. Theory* IT–21 (Mar. 1975), 228–230.

[5] M. R. Garey, "Optimal Binary Search Trees with Restricted Maximum Depth," *SIAM J. Comput.* 3 (June 1974), 101–110.

[6] S. W. Golomb, "Run-Length Encodings," *IEEE Trans. Inform. Theory* IT–12 (July 1966), 399–401.

[7] R. W. Hamming, *Coding and Information Theory*, Prentice-Hall, Englewood Cliffs, N.J., 1980.

[8] P. G. Howard & J. S. Vitter, "New Methods for Lossless Image Compression Using Arithmetic Coding," *Information Processing and Management* 28 (1992), 765–779.

[9] P. G. Howard & J. S. Vitter, "Practical Implementations of Arithmetic Coding," in *Image and Text Compression*, J. A. Storer, ed., Kluwer Academic Publishers, Norwell, Massachusetts, 1992, 85–112.

[10] P. G. Howard & J. S. Vitter, "Design and Analysis of Fast Text Compression Based on Quasi-Arithmetic Coding," *Information Processing and Management* 30 (1994), 777–794, also appears in shorter form in the proceedings of the Data Compression Conference, J. A. Storer and M. Cohn, eds., Snowbird, Utah, March 30-April 1, 1993, 98-107..

[11] P. G. Howard & J. S. Vitter, "Arithmetic Coding for Data Compression," *Proc. IEEE* 82 (June 1994), 857–865.

[12] P. G. Howard & J. S. Vitter, "Error Modeling for Hierarchical Lossless Image Compression," in *Proc. Data Compression Conference*, J. A. Storer & M. Cohn, eds., Snowbird, Utah, Mar. 24-26, 1992, 269–278.

[13] A. K. Huber, "A Hybrid Algorithm for Compression of Infrared Images of Space," Utah State University, M.S. Thesis, 1993.

[14] D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *Proceedings of the Institute of Radio Engineers* 40 (1952), 1098–1101.

[15] R. M. Neal, "Fast Arithmetic Coding Using Low-Precision Division," Unpublished manuscript, 1987.

[16] H. Printz & P. Stubley, "Multialphabet Arithmetic Coding at 16 MBytes/sec," in *Proc. Data Compression Conference*, J. A. Storer & M. Cohn, eds., Snowbird, Utah, Mar. 30-Apr. 1, 1993, 128–137.

[17] R. F. Rice, "Some Practical Universal Noiseless Coding Techniques," Jet Propulsion Laboratory, JPL Publication 79–22, Pasadena, California, Mar. 1979.

[18] J. J. Rissanen & K. M. Mohiuddin, "A Multiplication-Free Multialphabet Arithmetic Code," *IEEE Trans. Comm.* 37 (Feb. 1989), 93–98.

[19] D. C. Van Voorhis, "Constructing Codes with Bounded Codeword Lengths," *IEEE Trans. Inform. Theory* IT–20 (Mar. 1974), 288–290.

[20] I. H. Witten, R. M. Neal & J. G. Cleary, "Arithmetic Coding for Data Compression," *Comm. ACM* 30 (June 1987), 520–540.