Arithmetic Coding in Parallel

Jan Šupol and Bořivoj Melichar

Department of Computer Science & Engineering
Faculty of Electrical Engineering
Czech Technical University
Karlovo nám. 13, 121 35 Prague 2

e-mail: {supolj,melichar}@fel.cvut.cz

Abstract. We present a cost optimal parallel algorithm for the computation of arithmetic coding. We solve the problem in $\mathcal{O}(log\ n)$ time using $n/log\ n$ processors on EREW PRAM. This leads to $\mathcal{O}(n)$ total cost.

Keywords: arithmetic coding, NC algorithm, EREW PRAM, PPS, parallel text compression.

1 Introduction

There is still a need for data coding. The growing demand for network communication and for storage of data signals from space are not the only examples of coding needs. Many algorithms have been developed for text compression.

One of these is arithmetic coding [Mo98, Wi87], which is more efficient than the widely known Huffman algorithm [Hu52]. The latter rarely produces the best variable-size code, the arithmetic coding overcomes this problem. Arithmetic coding can be generated in $\mathcal{O}(n)$ time sequentially, and we present a well scalable NC parallel algorithm that generates the code in $\mathcal{O}(log\ n)$ time on EREW PRAM with $n/log\ n$ processors. This leads to $\mathcal{O}(n)$ total cost and a cost optimal algorithm.

Despite the large number of papers on the parallel Huffman algorithm (the last known [Lb99] is work optimal) there are only a few papers on parallel arithmetic coding. Most of these are based on a quasi-arithmetic coding [Ho92]. We know only two exceptions. The first [Yo98] is based on an N-processor hypercube and is not cost optimal. The second [Ji94] is mainly focused on the hardware implementation. Authors expected the processing speed of their tree-based parallel structure eight times as high as the speed of a sequential coder. This is still $\mathcal{O}(n)$ parallel time.

This paper is organized as follows. Section 2 provides a description of the sequential arithmetic coding algorithm. Section 3 presents some basic definitions. Section 4 describes the parallel prefix computation needed by our algorithm. Section 5 presents our parallel arithmetic coding algorithm. Section 6 describes the time complexity of our algorithm. Section 7 contains our conclusion. Note that this paper does not mention the decoding process.

2 Sequential Arithmetic Coding

First we review the sequential algorithm. Let $A = [a_0, a_1, \ldots, a_{m-1}]$ be the source alphabet containing m symbols and an associated set of frequencies $F = [f_0, f_1, \ldots, f_{m-1}]$ shows the occurrences of each symbol. Next we compute the array of probabilities $R = [r_0, r_1, \ldots, r_{m-1}]$ such that $r_i = f_i/T$ where $T = \sum_{i=0}^{m-1} f_i$, the array of high ranges $H = [h_0, h_1, \ldots, h_{m-1}]$ such that $h_i = \sum_{x=0}^{i} r_x$, the array of low ranges $L = [l_0, l_1, \ldots, l_{m-1}]$ such that $l_0 = 0$ and $l_i = h_{i-1}, i > 0$. Table 1 shows an example.

A	F	R	L	H
S	5	5/10=0.5	0.5	1.0
W	1	1/10=0.1	0.4	0.5
I	2	2/10=0.2	0.2	0.4
M	1	1/10=0.1	0.1	0.2
⊔	1	1/10=0.1	0.0	0.1

Table 1: Frequencies, probabilities and ranges of five symbols.

The string of symbols $S = [s^0, s^1, \ldots, s^{n-1}]$ is encoded as follows. The first character s^0 can be encoded by a number within an interval $[l_y, h_y)$ associated to a character $y = s^0, y \in A$. This notation $[a, b)$ means the range of real numbers from a to b, not including b. Let us define these two bounds as *LowRange* and *HighRange*.

As more symbols are input and processed, *LowRange* and *HighRange* are updated according to

$$LowRange^j = LowRange^{j-1} + (HighRange^{j-1} - LowRange^{j-1}) \times l_x,$$

$$HighRange^j = LowRange^{j-1} + (HighRange^{j-1} - LowRange^{j-1}) \times h_x,$$

where h_x and l_x are low and high ranges of new character $x \in A$, $LowRange^{-1} = 0$, $HighRange^{-1} = 1$. Table 2 indicates an example for the word "SWISS".

A	L&H	The calculation of low and high ranges
S	L	$0.0 + (1.0 - 0.0) \times 0.5 = 0.5$
	H	$0.0 + (1.0 - 0.0) \times 1.0 = 1.0$
W	L	$0.5 + (1.0 - 0.5) \times 0.4 = 0.70$
	H	$0.5 + (1.0 - 0.5) \times 0.5 = 0.75$
I	L	$0.7 + (0.75 - 0.7) \times 0.2 = 0.71$
	H	$0.7 + (0.75 - 0.7) \times 0.4 = 0.72$
S	L	$0.71 + (0.72 - 0.71) \times 0.5 = 0.715$
	H	$0.71 + (0.72 - 0.71) \times 1.0 = 0.720$
S	L	$0.715 + (0.72 - 0.715) \times 0.5 = 0.7175$
	H	$0.715 + (0.72 - 0.715) \times 1.0 = 0.7200$

Table 2: The process of arithmetic encoding.

3 Definitions

Our parallel algorithm is designed to run on the Parallel Random Access Machine (PRAM), which is a very simple synchronous model of the SIMD computer([Le92, Qu94, Tv94]). PRAM includes many submodels of parallel machines that differ from each other by conditions of access to the shared memory. Our algorithm works on the Exclusive Read Exclusive Write (EREW) PRAM model, which means that no two processors can access the same cell of the shared memory.

We define sequential time $SU(n)$ as the worst time of the best known sequential algorithm where n is the size of the input data. Parallel time $T(n,p)$ is the time elapsed from the beginning of a p-processor parallel algorithm solving a problem instance of size n until the last (slowest) processor finishes the execution.

Consider a synchronous p-processor algorithm A with $\tau = T(n,p)$ parallel steps. Let p_i be the number of processors active (working) at step $i \in \{1, 2, \ldots, \tau\}$ of A. Then the synchronous parallel work of A is

$$W(n,p) = T_1 + T_2 + \cdots + T_\tau.$$

Parallel cost (also called processor-time product) is defined as

$$C(n,p) = p \times T(n,p).$$

It is obvious that

$$SU(n) \leq W(n,p) \leq C(n,p).$$

If $SU(n) = W(n,p)$ then the algorithm is work optimal. If $SU(n) = C(n,p)$ then the algorithm is cost optimal.

The efficiency of the parallel algorithm is defined as

$$E(n,p) = \frac{SU(n)}{C(n,p)}.$$

Let E_0 be the constant such that $0 < E_0 < 1$. Then isoefficiency function $\psi_1(p)$ is the asymptotically minimum function such that

$$\forall n_p = \Omega(\psi_1(p)) : E(n_p, p) \geq E_0.$$

Hence, $\psi_1(p)$ gives asymptotically the lower bound on the instance size of a problem that can be solved by p processors with efficiency at least E_0.

Scalability is the ability to adapt itself to a changing number of processors or or to changing size of the input data. Good scalability means that if we want to use new processors we have to increase the size of our problem only a little. Fast growth of function ψ_1 provides poor scalability.

We say that class NC (Nick's class) is a set of algorithms that can be computed with at most polylogarithmic time and with at most a polynomial number of processors. These algorithms provide a high level of parallelization.

4 Parallel Prefix Computation

As far as our parallel algorithm is based on the parallel prefix algorithm we show how it works. The problem is defined as follows [La80]. Let $S = [s^0, s^1, \ldots, s^{n-1}]$ be the array of numbers. The prefix problem is to compute all the prefixes of the products

$$s^0 \otimes s^1 \otimes \cdots \otimes s^{n-1},$$

where $\otimes$ is an associative operation.

Fig. 1 shows the algorithm that assumes n processors $p^0, p^1, \ldots, p^{n-1}$ and array $M = [m^0, m^1, \ldots, m^{n-1}]$ of numbers stored in the shared memory. Every processor p^i also has a register y^i. From now on we will use EREW PRAM with similar conditions.

```
for i := 0, 1, ..., n − 1 do_in_parallel
   y^i := M[i];
for j := 0, 1, ..., ⌈log n⌉ − 1 do_sequentially
begin
   for i := 2^j, 2^j + 1, ..., n − 1 do_in_parallel
      y^i := y^i ⊗ M[i − 2^j];
   for i := 2^j, 2^j + 1, ..., n − 1 do_in_parallel
      M[i] = y^i;
end
```

Figure 1: Parallel prefix algorithm.

Fig. 2 indicates a parallel prefix algorithm computing an array of 7 numbers with the associative operation of sum. This is then called the parallel prefix sum.

Here we show the parallel time $T(n, p)$ of the parallel prefix computation on EREW PRAM. First we suppose that $p < n$. Each processor simulates n/p processors. This sequentially sums n/p numbers. This takes at most $4n/p$ steps (read first number, read second number, sum and write the result). After that the processors run the parallel prefix algorithm in time $\mathcal{O}(log\ p)$. So the parallel time, cost, efficiency and function ψ_1 take

$$T(n, p) = \mathcal{O}(n/p + log\ p),$$
$$C(n, p) = \mathcal{O}(n + p\ log\ p),$$
$$E(n, p) = \mathcal{O}(\frac{n}{n + p\ log\ p}),$$
$$\psi_1(p) = \mathcal{O}(p\ log\ p).$$

We can say that the parallel prefix algorithm is a well scalable NC algorithm due to the definitions in Section 3. If $p = n$ then

$$T(n, n) = \mathcal{O}(n/n + log\ n) = \mathcal{O}(log\ n),$$
$$C(n, n) = \mathcal{O}(n + n\ log\ n) = \mathcal{O}(n\ log\ n).$$

However, when $p = n/log\ n$ then

$$T(n, n/log\ n) = \mathcal{O}(n\ log\ n/n + log\ n - log\ log\ n) = \mathcal{O}(log\ n),$$
$$C(n, n/log\ n) = \mathcal{O}(n + n/log\ n(log\ n - log\ log\ n)) = \mathcal{O}(n).$$

Hence, we have obtained a parallel cost optimal algorithm.

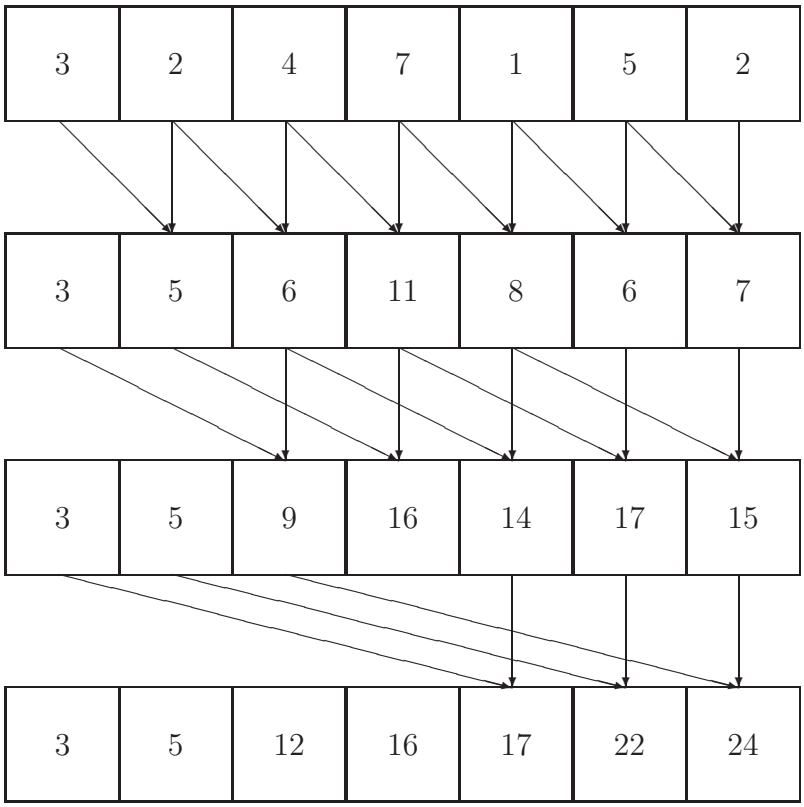

Figure 2: Parallel prefix sum example.

5 Parallel Arithmetic Coding

Recall that we use the array $A = [a_0, a_1, \ldots, a_{m-1}]$ of the source alphabet containing m symbols, the associated set of frequencies $F = [f_0, f_1, \ldots, f_{m-1}]$, the associated set of probabilities $R = [r_0, r_1, \ldots, r_{m-1}]$ so that $r_i = f_i/T$ where $T = \sum_{i=0}^{m-1} f_i$, the array of low ranges $L = [l_0, l_1, \ldots, l_{m-1}]$, the array of high ranges $H = [h_0, h_1, \ldots, h_{m-1}]$ so that $l_0 = 0$, $l_i = h_{i-1}, i > 0$ and $h_i = l_i + r_i$.

Our idea of parallelism is that we have a string $S = [s^0, s^1, \ldots, s^{n-1}]$ of n characters to encode. Each processor p^j is associated with a character s^j and computes variables *LowRange* and *HighRange* for that character.

5.1 Preliminaries

We suppose that we have an array $Range = [range^0, range^1, \ldots, range^{n-1}]$ for our algorithm. Each $range^j$ is initialized with probability r_y such that $a_y = s^j$ where j is the index of the j-th character in the input string and $s^j \in A$. We also suppose that we have an array $Low = [low^0, low^1, \ldots, low^{n-1}]$. Each low^j is initialized with value l_y such that $a_y = s^j$. We need at least one variable $high$ initialized with value h_y such that $a_y = s^{n-1}$.

5.2 Changes in Sequential Algorithm

Let us return to sequential arithmetic coding and try to change the algorithm a bit so that it can be parallelized. Recall the bounds computation

$$LowRange^j = LowRange^{j-1} + (HighRange^{j-1} - LowRange^{j-1}) \times l_x,$$

$$HighRange^j = LowRange^{j-1} + (HighRange^{j-1} - LowRange^{j-1}) \times h_x,$$

where h_x and l_x are low and high ranges of new character $x \in A$, $LowRange^{-1} = 0$, $HighRange^{-1} = 1$ and mark the cumulative lower and higher bounds

$$LR^j = (HighRange^{j-1} - LowRange^{j-1}) \times l_x,$$

$$HR^j = (HighRange^{j-1} - LowRange^{j-1}) \times h_x.$$

So the values *LowRange* and *HighRange* are updated as

$$LowRange^j = LowRange^{j-1} + LR^j,$$

$$HighRange^j = LowRange^{j-1} + HR^j$$

and we now focus only on the variables LR and HR now.

$$\begin{aligned} LR^j &= (HighRange^{j-1} - LowRange^{j-1}) \times l_x = \\ &= (LowRange^{j-2} + HR^{j-1} - LowRange^{j-2} - LR^{j-1}) \times l_x = \\ &= (HR^{j-1} - LR^{j-1}) \times l_x, \end{aligned}$$

$$\begin{aligned} HR^j &= (HighRange^{j-1} - LowRange^{j-1}) \times h_x = \\ &= (LowRange^{j-2} + HR^{j-1} - LowRange^{j-2} - LR^{j-1}) \times h_x = \\ &= (HR^{j-1} - LR^{j-1}) \times h_x. \end{aligned}$$

Moreover, $LowRange^j$ can be computed as

$$\begin{aligned} LowRange^j &= LR^j + LowRange^{j-1} = LR^j + LR^{j-1} + LowRange^{j-2} = \\ &= \cdots = LR^j + LR^{j-1} + \cdots + LR^0 + LowRange^{-1} = \\ &= \sum_{x=0}^{j} LR^x + LowRange^{-1} = \sum_{x=0}^{j} LR^x \end{aligned}$$

because $LowRange^{-1} = 0$.

The change in our algorithm is that we first compute the cumulative lower and higher bounds and next we simply compute the sum of these cumulative bounds so that we obtain the final bounds *LowRange* and *HighRange*.

Let us see how the variables LR and HR can be computed for the word "SWISS". We declare that LR^0 is the LR variable for the first character $s^0 =$ "S'' and l_x, h_x, r_x are lower range, higher range and probability of character $x \in A$. LR^{-1} and HR^{-1} are initial cumulative bounds for a number that represents the encoded text S. For arithmetic coding this number is defined by default as an interval [0,1). That is why $LR^{-1} = LowRange^{-1} = 0$ and $HR^{-1} = HighRange^{-1} = 1$.

$LR^{-1} = 0$
$HR^{-1} = 1$
$LR^0 = (HR^{-1} - LR^{-1}) \times l_s = 1.0 \times 0.5 = 0.5$
$HR^0 = (HR^{-1} - LR^{-1}) \times h_s = 1.0 \times 1.0 = 1.0$
$LR^1 = (HR^0 - LR^0) \times l_w = (h_s - l_s) \times l_w = r_s \times l_w = 0.5 \times 0.4 = 0.2$
$HR^1 = (HR^0 - LR^0) \times h_w = (h_s - l_s) \times h_w = r_s \times h_w = 0.5 \times 0.5 = 0.25$
$LR^2 = (HR^1 - LR^1) \times l_i = (r_s \times h_w - r_s \times l_w) \times l_i = r_s \times r_w \times l_i = 0.5 \times 0.1 \times 0.2 = 0.01$
$HR^2 = (HR^1 - LR^1) \times h_i = (r_s \times h_w - r_s \times l_w) \times h_i = r_s \times r_w \times h_i = 0.5 \times 0.1 \times 0.4 = 0.02$
$LR^3 = (HR^2 - LR^2) \times l_s = (r_s \times r_w \times h_i - r_s \times r_w \times l_i) \times l_s = r_s \times r_w \times r_i \times l_s = 0.005$
$HR^3 = (HR^2 - LR^2) \times h_s = (r_s \times r_w \times h_i - r_s \times r_w \times l_i) \times h_s = r_s \times r_w \times r_i \times h_s = 0.01$
...

So it is obvious that the lower bound of the j-th character LR^j and the higher bound of the j-th character HR^j can be computed as

$$LR^j = (\prod_{x=0}^{j-1} r^x) \times l^j, j > 0,$$

$$HR^j = (\prod_{x=0}^{j-1} r^x) \times h^j, j > 0.$$

5.3 Parallel Prefix Production

```
for i := 0, 1, ..., n − 1 do_in_parallel
   y^i := Range[i];
for j := 0, 1, ..., ⌈log n⌉ − 1 do_sequentially
begin
   for i := 2^j, 2^j + 1, ..., n − 1 do_in_parallel
      y^i := y^i × Range[i − 2^j];
   for i := 2^j, 2^j + 1, ..., n − 1 do_in_parallel
      Range[i] = y^i;
end
```

Figure 3: Parallel prefix production algorithm.

These new LR and HR variables are exactly what we need, because $\prod_{x=0}^{j-1} r^x$ can be computed in parallel as we immediately show. Computation of $\prod_{x=0}^{j} r^x = \prod_{x=0}^{j} range^x$ can be done by the parallel prefix production algorithm explained in Section 4, as shown in Fig. 3. Table 3 indicates the parallel prefix algorithm in our example for the word "SWISS".

S	W	I	S	S
0.5	0.1	0.2	0.5	0.5
0.5	0.05	0.02	0.1	0.25
0.5	0.05	0.01	0.005	0.005
0.5	0.05	0.01	0.005	0.0025

Table 3: Parallel prefix production example for the word "SWISS".

5.4 Cumulative Bounds Computation

If we have computed $\prod_{x=0}^{j-1} r^x$ we can obtain the variables LR^j and HR^j simply as the product of $\prod_{x=0}^{j-1} r^x \times l^j$ and $\prod_{x=0}^{j-1} r^x \times h^j$. Parallel algorithm computing the variables LR and the variable HR^{n-1} is shown in Fig. 4. The variables HR are not exactly needed, except for the last one HR^{n-1}. If these variables are required, they can be computed in a similar way. The value HR^{n-1}, which is the cumulative high range, is computed after the parallel prefix production computation as

$$HR^{n-1} = (\prod_{x=0}^{n-2} r^x) \times h^{n-1}.$$

Table 4 shows this computation in our example for the word "SWISS". Note that the results correspond to the cumulative bounds in our sequential example.

```
do_sequentially
begin
   y^{n-1} := High;
   y^{n-1} := y^{n-1} × Range[n − 2];
   High := y^{n-1};
   y^0 := 1;
end
for i := 1, 2, . . . ,n − 1 do_in_parallel
   y^i := Range[i − 1];
for i := 0, 1, . . . ,n − 1 do_in_parallel
begin
   y^i := y^i × Low[i];
   Low[i] := y^i;
end
```

Figure 4: Parallel computation of the variables LR and HR^{n-1}.

Now we have computed the cumulative high and low ranges. The array *Low* contains the LR values and the field *High* contains the value HR^{n-1}. Next we have to compute the sum of these cumulative ranges LR so that we shall obtain the required bounds *HighRange* and *LowRange* for arithmetic compression of string S.

L/H	S	W	I	S	S
LR	0.5	0.2	0.01	0.005	0.0025
HR	1	0.25	0.02	0.01	0.005

Table 4: Low and high ranges.

5.5 Computation of Low and High Ranges

In Section 5.4 we computed the cumulative bounds *LR* and *HR*. Here we show how to obtain the bounds earlier declared as *LowRange* and *HighRange* for the compressed text. These values can be computed as shown in Section 5.2 as

$$LowRange^j = (\sum_{x=0}^{j-1} LR^x) + LR^j,$$

$$HighRange^j = (\sum_{x=0}^{j-1} LR^x) + HR^j.$$

To compute the sum we can use the parallel prefix algorithm once more, exactly the parallel prefix sum shown in the former text. Finally, after computing the sum, the variable $HighRange^{n-1}$ is obtained as

$$HighRange^{n-1} = LowRange^{n-2} + HR^{n-1}.$$

This algorithm is shown in Fig. 5. The array *Low* contains the values *LowRange* and the field *High* contains the value $HighRange^{n-1}$. Our example for the word "SWISS" is shown in Table 5.

```
for i := 0, 1, ..., n − 1 do_in_parallel
   y^i := Low[i];
for j := 0, 1, ..., ⌈log n⌉ − 1 do_sequentially
begin
   for i := 2^j, 2^j + 1, ..., n − 1 do_in_parallel
      y^i := y^i + Low[i − 2^j];
   for i := 2^j, 2^j + 1, ..., n − 1 do_in_parallel
      Low[i] = y^i;
end
do_sequentially
begin
   y^{n−1} := High;
   y^{n−1} := y^{n−1} + Low[n − 2];
   High := y^{n−1};
end
```

Figure 5: *LowRange* and $HighRange^{n-1}$ computation algorithm.

S	W	I	S	S
0.5	0.2	0.01	0.005	0.0025
0.5	0.7	0.21	0.015	0.0075
0.5	0.7	0.71	0.715	0.2175
0.5	0.7	0.71	0.715	0.7175

Table 5: Parallel prefix sum example.

6 Time and Cost Complexities

Our algorithm does not say how to set the arrays *Range*, *Low* and the variable *High* in a preliminary phase. However, having set the arrays A, R, L and H, this can be done in time $\mathcal{O}(1)$ on CREW PRAM with a good hash function that returns an index in the array A of an input character from the input string S.

Our EREW PRAM algorithm consists of three phases. In the first phase, the parallel prefix production is computed. As shown in Section 4, this can be done in time $\mathcal{O}(n/p + log\ p)$ where p is the number of used processors and n is the size of the input. In the second phase, shown in Fig. 4, we have computed the cumulative bounds LR and HR in time $\mathcal{O}(n/p)$. The third phase, the parallel prefix sum shown in Fig. 5, also takes $\mathcal{O}(n/p + log\ p)$ time. The computation of $HighRange^{n-1}$ takes only $\mathcal{O}(1)$ time in any phase. So the time and cost of our algorithm are

$$T(n,p) = \mathcal{O}(n/p + log\ p),$$

$$C(n,p) = \mathcal{O}(n + p\ log\ p).$$

If $p = n/log\ n$ then the total time is $\mathcal{O}(log\ n)$ and the cost is $\mathcal{O}(n)$.

Because our algorithm consists mainly of parallel prefix computation, it inherits its best properties. Our algorithm is therefore a well scalable NC algorithm, and it can be implemented as the cost optimal algorithm.

7 Conclusions

We have presented a parallel NC algorithm for computation of arithmetic coding. We have solved the problem in $\mathcal{O}(log\ n)$ time using $n/log\ n$ processors on EREW PRAM. Our algorithm leads to $\mathcal{O}(n)$ total cost and is cost optimal.

The preliminary phase is a weakness of our algorithm. However, if we were able to construct a good adaptive parallel arithmetic coding based on our algorithm, it could solve this problem.

Another question is how to make a good parallel arithmetic decoding algorithm.

References

[Ho92] Howard, Paul G., Jeffrey Scott Witter (1992): Parallel Losseless Image Compression Using Huffman and Arithmetic Coding. Proceedings of the IEEE Data Compression Conference, 299-308.

[Hu52] Huffman, David (1952): A method for the construction of minimum-redundancy codes. Proceedings of the Inst. Radio Engineers, **40**: 1098-1101.

[Ji94] Jiang J., S. Jones (1994): Parallel design of arithmetic coding. IEE Proceedings-Computers and Digital Techniques, **141**(6):327-333, November.

[La80] Ladner, Richard and Michael J. Fisher (1980): Parallel Prefix Computation. Journal of the ACM, **27**(4):831-838, October.

[Lb99] Laber, Eduardo Sany, Ruy Luiz Milidi and Artur Alves Pessoa (1999): A Work Efficient Parallel Algorithm for Constructing Huffman Codes. Proceedings of the IEEE Data Compression Conference DCC'99.

[Le92] Lewis, T.G. and H. El-Rewini (1992): Introduction to Parallel Computing. Prentice Hall.

[Qu94] Quinn, M.J.(1994): Parallel Computing Theory and Practise. McGraw-Hill.

[Mo98] Moffat, Alistar, Redford Neal, and Ian H.Witten (1998): Arithmetic Coding Revisited. ACM Transactions on Information Systems, **16**(3):256-294, July.

[Tv94] Casavant, T.L., P. Tvrdík and F. Plášil, editors (1994): Parallel Computers: Architectures, Languages, and Algorithms. IEEE CS Press.

[Wi87] Witten, Ian H., Redford Neal and John G. Cleary (1987): Arithmetic coding for Data Compression. Communications of the ACM **30**(6):520-540.

[Yo98] Youssef A. (1998): Parallel Algorithms for Entropy Coding Techniques. Proceedings of European Parallel and Distributed Systems. ACTA Press.