SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2002; **32**:1223–1237 (DOI: 10.1002/spe.480)

Context-based compression of binary images in parallel

Eugene Ageenko[1,*,†], Martti Forsell[2] and Pasi Fränti[1]

[1]*Department of Computer Science, University of Joensuu, PB 111, FIN-80101 Joensuu, Finland*
[2]*VTT Electronics, PB 1100, FIN-90571 Oulu, Finland*

SUMMARY

Binary images can be compressed efficiently using context-based statistical modeling and arithmetic coding. However, this approach is fully sequential and therefore additional computing power from parallel computers cannot be utilized. We attack this problem and show how to implement the context-based compression in parallel. Our approach is to segment the image into non-overlapping blocks, which are compressed independently by the processors. We give two alternative solutions about how to construct, distribute and utilize the model in parallel, and study the effect on the compression performance and execution time. We show by experiments that the proposed approach achieves speedup that is proportional to the number of processors. The work efficiency exceeds 50% with any reasonable number of processors. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: image compression; context modeling; JBIG; parallel algorithms; EREW; PRAM

INTRODUCTION

Binary images can be efficiently compressed using context-based statistical modeling and arithmetic coding. The latest international compression standards, JBIG [1] (*Joint Bi-level Image Group*) and JBIG2 [2,3] are both based on this approach. The methods underlying these standards process the image pixel-by-pixel using backward-adaptive modeling, in which the compression of a single pixel depends on all the previous pixel data. Moreover, the bit stream provided by arithmetic coding is unbreakable and the compression is therefore completely sequential in its nature.

The images in question are binary images such as text documents and engineering drawings, or images composed of a limited number of binary layers such as topographic maps. The images of this kind have typically only a few color tones but high spatial resolution to represent fine details

*Correspondence to: Eugene Ageenko, Department of Computer Science, University of Joensuu, PB 111, FIN-80101 Joensuu, Finland.
†E-mail: ageenko@cs.joensuu.fi

Contract/grant sponsor: National Technology Agency of Finland (TEKES); contract/grant number: 954/401/99

Published online 19 September 2002

Received 23 July 2001
Revised 24 April 2002
Accepted 23 May 2002

such as text and graphics objects. The image sizes can be huge. A single-page fax image is about 1000×2000 pixels, while a typical topographic image is about 5000×5000 pixels. The latter image can be compressed usually by a factor 10:1 or more, which corresponds to the file size reduction from 3 megabytes to 300 kilobytes. The compression of a single pixel is very fast consisting of an index construction, table look-up and a few bit-level operations of the arithmetic coder without any multiplications. However, the compression time of images containing millions of pixels will easily reach several minutes. The compression speed can degrade the usability of the compression in on-line applications.

Although, the research toward parallel computers is still far from being complete, it is inevitable that parallel computers will eventually take the place of the current sequential computers in everyday life. For example, let us suppose that we have a workstation with 10 processors instead of only one. The question is whether we can utilize the extra computing power when compressing and decompressing the images. It is certain that the user would greatly appreciate it if the image were decompressed within a second or two instead of 10 or 20 seconds, if there were the computing potential to do so.

Parallel Random Access Machine (PRAM) is a general model for parallel computation first proposed by Fortune and Wyllie [4]. The model assumes that the communication between processors comes at no cost. Therefore, PRAM can be viewed as a virtual parallel machine and serve as a design model for the development of very fast computational algorithms. Although the PRAM model has previously been considered somewhat idealistic, there has been a considerable effort to simulate PRAM on a physically distributed memory machine consisting of processors and memory modules interconnected through a communication network [5–10]. A few experimental and commercially available PRAM-style computers have already been designed, including the Saarbrücken Parallel Random Access Machine [5], Cray MTA supercomputer [7], previously known as the Tera MTA, and 64 SB-RAM [9].

In general, computational problems can be divided into three types: strictly sequential; apparently sequential; and parallel. In strictly sequential portions, e.g. the IIR-filter, potentially every statement depends on the result of the preceding statement, and there is no way to implement them in parallel. In apparently sequential portions, e.g. the prefix sum, potentially every statement in the algorithm depends on the result of the preceding statement but the computation and control can be rearranged so that a parallel algorithm can be designed. In parallel portions, e.g. the matrix sum, the number of dependencies between statements is so low that a parallel algorithm can be designed quite easily with the PRAM model.

With respect to the previous classification, context-based compression contains strictly sequential portions, namely the adaptive modeling and arithmetic coding are such processes. Thus, the backward-adaptive compression cannot be implemented under a parallel model such as PRAM. In this paper, we attack the above problem and present a method for the context-based compression of binary images in parallel. Our approach is to divide the compression problem into several smaller sub-problems by segmenting the image into $b \times b$ non-overlapping rectangular blocks. The blocks are processed independently on parallel processors, and are stored in the same file in a sequential manner assuming that the file I/O is not a bottleneck. The index table is stored in the file header to locate the starting points of the code blocks. In this way, decompression can also be performed in parallel. Moreover, the JBIG2 file format supports this kind of file organization, where the image is divided into separate blocks.

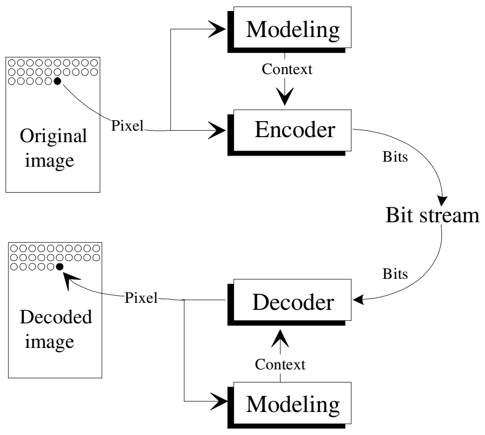

Figure 1. Block diagram of backward-adaptive context-based compression.

A drawback of the block decomposition is the decrease in compression performance due to the increased learning cost. To alleviate this problem, we consider a forward-adaptive variant of the statistical modeling [11], in which the blocks share the same initial statistical model optimized for the entire image. We will give solutions about how the construct, distribute and utilize the model in parallel, and study the effect on the compression performance and execution time.

In the following sections we briefly recall the context-based compression, describe image compression using a parallel random access machine, discuss the implementation details, and perform the experimental study.

CONTEXT-BASED COMPRESSION

In context-based compression, an image is processed pixel-by-pixel in raster-scan order starting from the top leftmost pixel of the image as shown in Figure 1. The probabilities of the black and white pixels are conditioned on the context determined by the combination of the already coded neighboring pixels. A three-line ten-pixel context template is shown in Figure 2. Both the encoder and decoder estimate the model dynamically during the compression and decompression, respectively. The estimation starts from scratch and adapts the model to the input data.

The approach is adopted in the international standard JBIG [1] and the emerging standard JBIG2 [2,3]. Unlike JBIG, JBIG2 segments a page into different classes of image data, in particular, textual, halftone and generic (other), and utilize the repetitive nature of the textual and halftone images. However, the encoding of the data other than text or halftones remains similar to JBIG with the difference that a newer version of the coder is used, namely the MQ-coder.

The MQ-coder is an approximate implementation of arithmetic coding tailored for binary data. Its pre-ancestor, Q-coder, has similar working principles and is introduced in [12]. The coder

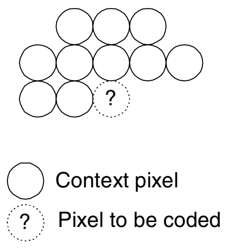

Figure 2. Example of a template of 10 pixels resulting in 1024 contexts.

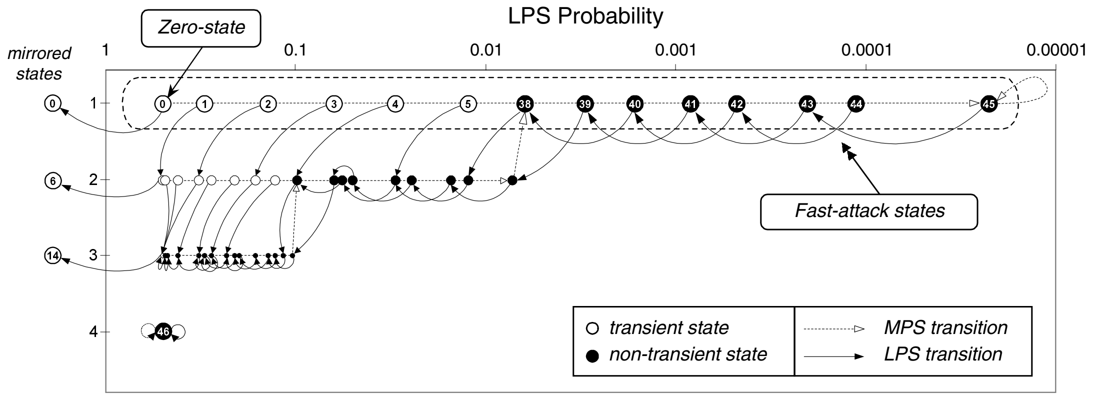

Figure 3. Spatial organization of the state automaton in MQ-coder and the transition sketch for the fast-attack states. Because of the mirror symmetry regarding the change in sense of LPS and MPS, only half of the states are depicted.

sub-optimality is compensated by the sophisticated automaton-based probability estimation, providing fast adaptation to the source data. Instead of maintaining pixel counts, the estimation process is implemented as a state automaton consisting of 94 states (the QM-coder that is used in JBIG has 226 states). The automaton is a Markov chain containing one state for each probability estimate. The states are organized in rows that are ordered by the level of adaptation. The automaton has mirror symmetry about the change in the sense of more-probable symbol (MPS) color, and we therefore consider only 47 states, see Figure 3.

The statistical model is usually integrated with the coder. It consists of the pointers to the automaton states, one for each context. The adaptation process starts from the nearly uniform model represented by the zero state. After the pixel is coded, the model is updated by assigning a new, updated, state index for the pixel context.

The probability estimation is derived from the arithmetic coder renormalization and is based on the Bayesian estimation concept [13]. The renormalization of the coding interval occurs always after a less-probable symbol (LPS), and if necessary, after a MPS symbol. After each MPS renormalization, the automaton makes the transition to the next state situated to the right in the same row, having a smaller LPS probability. After each LPS renormalization, a transition is made to the state with a larger LPS probability, which is the appropriate state in the row at the next level in case of the *transient* state, or to the preceding state in the same row in case of *non-transient* states. Transient states are, therefore, visited only during the learning stage, and the pointers stabilize eventually to the non-transient states. If the statistics change later, the non-transient states can be re-entered from other non-transient states, making local adaptation possible.

PARALLEL COMPRESSION

Several attempts have been reported in the literature concerning the parallelization of lossless data compression. Henriques and Ranganathan [14] have proposed a systolic array-based architecture for implementing the Ziv–Lempel algorithm in parallel. Howard and Vitter [15] have proposed parallel algorithms using hierarchical modeling with Huffman and quasi-arithmetic coding. Their method assumes static prediction-based modeling scheme, which is applicable for the lossless compression of gray-scale images but not necessarily for binary images. They use the PRAM model like us, but assume a stronger variant capable of concurrent reading and limited concurrent writing.

For context-based binary image compression, efforts have mainly been focused on low-level and hardware-oriented parallelization. For example, Horie *et al.* [16] and Tarui *et al.* [17] have both proposed special purpose hardware in order to implement JBIG faster. These methods are based on pipelining the execution logic, i.e. they apply instruction-level parallelism, which cannot be generalized to a large number of processors. Jiang [18] has focused on the parallelization of the adaptive binary arithmetic coding. Only a marginal speedup can be achieved at the instruction level due to the strictly sequential nature of the problem. In theory, the idea generalizes to any number of bits to be processed in parallel but at the cost of exponentially increasing complexity of the implementation, and at the cost of decreasing compression performance.

To sum up, the existing approaches cannot be generalized to a massive parallelism without significant problems. Boliek *et al.* [19] have recognized the problem and proposed a general solution with a different approach; divide the input into separate data streams to be compressed independently. They identify the problems that must be solved to implement such parallelization but they do not give a practical solution. In the following, we will carry on this approach and introduce an efficient parallel counterpart to context-based binary image compression. The method is executable in a physically feasible general-purpose parallel computer with a wide range of the number of processors.

EREW PRAM model

PRAM [4] is a general model for parallel computations. It consists of P sequential processors, each having a private, or local, memory for its own computation. All processors are connected to a shared random access memory (see Figure 4). All communication between the processors is done via the shared memory. All processors execute the same program synchronously, but a processor may branch within the program independently of other processors. The PRAM model has several different variants according to the level of shared memory access [20]. In this paper, we refer to just one of them, to

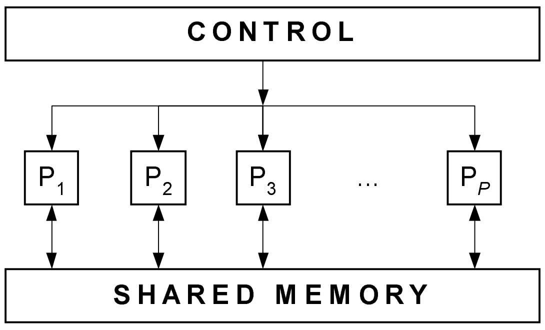

Figure 4. The PRAM model. $P_1, \ldots, P_P$ are processors.

exclusive read exclusive write (EREW). In this model, processors are not allowed to read or write to the same memory location simultaneously. Most of the other models can be treated analogously.

We assume that the PRAM model provides us with the following tools. The first tool is the FOR...PARDO loop-statement, which executes the commands inside the loop synchronously in parallel. Using this command, a simple loop consisting of n iterations and not accessing the same memory locations can be performed on n processors in $O(1)$ time in comparison to the $O(n)$ time required by sequential processing. The second tool is the *ParSum*() routine for computing the sum of n numbers. Because of the limitations of EREW, it requires $O(\log n)$ time. The third tool is the datum broadcast routine *Distribute*(), which distributes a single datum to n processors in $O(\log n)$ time.

Parallel implementation

The proposed parallel implementation (referred to here as P-JBIG) is based on the decomposition of the image into $B = b \times b$ equally sized blocks. A processor dedicated to the block compresses each block separately providing a simple division of the problem into parallel sub-problems, see Figure 5. Furthermore, the block index table indicating the location of the block in the compressed file can be constructed. It provides direct access to the compressed image file, and therefore enables efficient and independent decompression of a particular block.

For simplicity, we will assume that the number of blocks B equals the number of processors P. If this is not practical, it is possible to use standard parallel algorithmic methods to handle the situation. If the number of processors is greater than the number of blocks then we de-allocate additional processors for other purposes. On the other hand, if the number of processors is smaller than the number of blocks then more than one block is processed with a single processor. According to the well-known Brent's theorem [21], this processor allocation can be performed without sacrificing the work optimality of the algorithm.

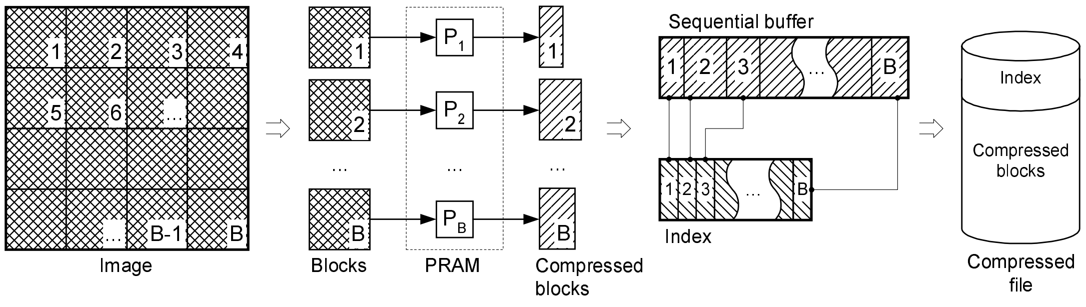

Figure 5. Parallelization using block decomposition and indexing.

FORWARD-ADAPTIVE VARIANT

The block decomposition has the effect that there are fewer pixels to be coded by a single processor. This means that the backward-adaptive modeling has less time to adapt to the statistics of the image, which increases the learning cost. Another problem related to the compression of small blocks is the inefficiency near block boundaries. The pixels located outside the block cannot be used in the context template, which deteriorates the compression performance. Previous studies [11] indicate that the compression inefficiency remains tolerable if the block size is kept to 256×256 pixels or higher.

In order to compensate for the compression inefficiency, we consider a forward-adaptive variant of the parallel compression. The proposed method is based on the ideas presented in [11], and is referred here as the parallel forward-adaptive method (P-FAM). The method makes two passes over the image: modeling and actual compression. The sketch of the P-FAM is outlined in Figure 6, and will be detailed in the following.

Modeling

During the modeling pass, the input image is analyzed and the initial model is constructed. First, statistics are collected from the image by summing up the number of white and black pixels. Each processor handles a single block using its own counter (Loop 1). After the blocks have been processed, the statistics for each of M contexts are summed in parallel into the first column of the counter tables (Loop 2). In Loop 3, statistics are translated to the nearest *fast-attack states* of the MQ-coder (states numbered 0–5 and 38–45 and their mirrored states). Finally, the resulting model (states) is distributed to every processor (Loop 4). The distribution of the model is necessary due to the access limitations of the EREW PRAM model.

Coding and decoding

During the coding pass (Loop 5), the blocks are compressed essentially in the same way as if using P-JBIG method. The difference is that the MQ-coder is initialized not to the blank model (zero state),

```
{LOOP 1: Analyse image}
FOR p := 1 TO P PARDO BEGIN
  FOR (each pixel x of block p) DO BEGIN
    c := DetermineContext (x);
    total[c, p] := total[c, p] + 1;
    IF x = white THEN whites[c, p] :=  whites[c, p] + 1;
  END;
END;

{LOOP 2: Sum the statistics of blocks}
FOR  m := 1 TO M PARDO BEGIN
  total[m, 1] := ParSum (total[m, 1..P]) ;
  whites[m, 1] := ParSum (whites[m, 1..P]) ;
END

{LOOP 3: Construct model}
FOR  m := 1 TO M PARDO BEGIN
  state[m, 1] := SelectNearestState (whites[m, 1] /  total[m, 1] ) ;
END

{LOOP 4: Spread the model to the processors}
FOR  m := 1 TO M PARDO BEGIN
  Distribute(state[m, 1],  state[m, 2..P]);
END

{LOOP 5: Compress image}
FOR p := 1 TO P PARDO BEGIN

  {LOOP 5.1: Initialize the models}
  FOR m := 0 TO M DO BEGIN
    InitializeModels (m, state[m, p]);
  END;

  {LOOP 5.2: Compress the blocks}
  FOR (each pixel x of block p) DO BEGIN
    c := DetermineContext (x);
    state[c, p] := EncodePixel (x, state[c, p]);
  END;
END;
```

Figure 6. The sketch of the P-FAM algorithm.

but to the model constructed in the previous stage. The initialization (Sub-loop 5.1) takes the context number (*m*) and the state index (*state*) as input and restores the respective variables for the appropriate context in the MQ-coder. The encoding function *EncodePixel*() takes pixel color and the model and returns the updated model.

Decompression is similar to the compression, except that the model is read from the compressed file, eliminating the need for an extra pass over the image. If only a part of an image is decoded, the number of processors required equals the number of blocks that contains the portion of the required image.

Table I. The computational complexity of the P-FAM algorithm. $N \equiv P$ is assumed.

Loop	Parallel time
1. Analysis	$O(N/P)$
2. Statistics summation	$O(M + \log_2 \max\{1, P/M\})$
3. Model construction	$O(\max\{1, M/P\})$
4. Model redistribution	$O(M + \log_2 \max\{1, P/M\})$
5.1 Model restoring	$O(M)$
5.2 Coding	$O(N/P)$
Total	$O(N/P) + O(M + \log_2 \max\{1, P/M\})$

Execution time analysis

According to [8], it is realistic to simulate EREW PRAM on a parallel computer with multithreaded processors and a coated mesh communication network in $O(1)$ time. Recent experiments with a prototype 64 SB-PRAM machine featuring the even stronger CRCW PRAM model have shown an overhead of only about 0.7% between runtime on the real hardware and the theoretical model [9]. The I/O cost is ignored in the following analysis.

The time complexity of the sequential algorithm (JBIG) is $O(N)$. The asymptotic estimations of the complexity of the different loops of P-FAM are summarized in Table I. The analysis for most loops of the algorithm is trivial. The image analysis step (Loop 1) requires $O(N/P)$ time, as there are N pixels to be processed by P processors. The mapping (Loop 3) takes $O(M/P)$ respectively, because there are M models to be constructed. The model initialization (Loop 5.1) takes $O(M)$ because it must be performed by every processor. The encoding step (Loop 5.2) requires $O(N/P)$ time in the same way as the analysis step.

The Loops 2 and 4 are analyzed as follows. All P processors are utilized for these loops. The summation of the statistics requires approximately $\log_2 P$ iterations. On the ith iteration, P/i columns of the length M are summed in parallel using 2^i processors for each column. Therefore, the complexity totals to

$$O(M/2) + O(M/4) + \cdots + O(M/P) = O(M)$$

The above equation stands only when P is smaller than or equal to M. If $P > M$, the equation becomes

$$\begin{aligned} O(M/2) + O(M/4) + \cdots + O(M/M) + O(1) + \cdots + O(1) &= O(M) + O(\log_2 P) - O(\log_2 M) \\ &= O(M + \log_2\{P/M\}) \end{aligned}$$

The difference $\lceil \log_2 P \rceil - \lceil \log_2 M \rceil$ is the number of $O(1)$'s in the above sum. In general, the complexity of the Loop 2 (and similarly Loop 4) amounts to

$$O(M + \log_2 \max\{1, P/M\})$$

The total complexity of P-FAM is

$$O(N/P) + O(M + \log_2 \max\{1, P/M\})$$

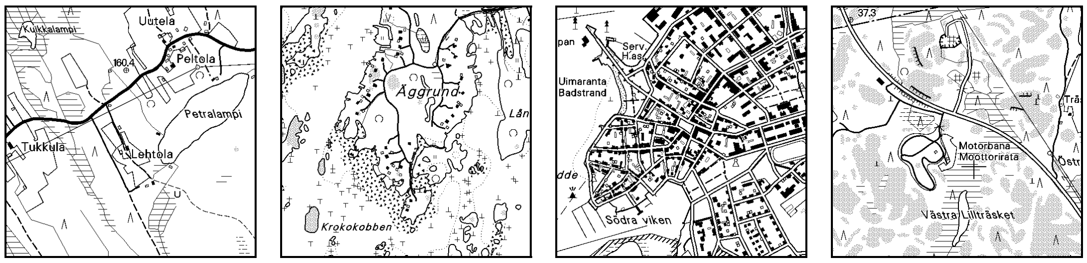

Figure 7. Sample fragments of the test images. All fragments of the size 500 × 500 pixels. Reproduced by permission of the National Land Survey of Finland.

For large image sizes and a relatively low number of blocks (processors), the P-FAM is asymptotically P times faster than JBIG. The relative speedup of the parallel implementation regresses when the number of processors increases. This regress is because the term $\log_2(P/M)$ starts to dominate the time complexity for larger values of P. It only has meaning though, as the logarithmic growth is very slow in comparison to the overall benefit of the parallelization. For example, in our experiments we have considered values no higher than $M = 1024$ and $P_{\mathrm{MAX}} = 10\,000$.

EXPERIMENTS

The compression performance and execution speed of sequential and parallel implementations are demonstrated by compressing a set of GIS Topographic images (see Figure 7). For testing, we use a PC based on a Pentium-III 600 MHz with RAM-bus architecture, running Windows 2000 Pro. We exclude file I/O and memory allocation calls from the time computation. In the code size calculation, we exclude the overhead of the file and include only pure codestream sizes together with the indices and the initial model, when applicable. We repeat the experiments for a different number of parallel processors P. We partition the image into $b \times b$ blocks and assume that $P = b^2$. The parameter b is varied from 1 to 100 resulting in 1 to 10 000 processors.

Execution time

The execution time simulation is performed using a sequential implementation of the algorithm. The actions taken to simulate various stages of the algorithm are illustrated in Table II. The compression of some block may take longer than the compression of another. It is caused by the different complexity of the data in different blocks. Since the PARDO loop is always synchronized, the time required to compress all blocks in parallel equals the time required to compress the block of the highest complexity. We only present the compression times here because coding and decoding processes are symmetrical.

At steps 2 and 4, we also take into account the simulation cost as measured using the forthcoming on-chip PRAM realization presented in [10]. It takes into account all the practical aspects of EREW PRAM simulation such as multithreading in processors, communication and routing, and low-level

Table II. Estimation of the execution times using sequential processing.

Loop	Used in P-JBIG	Used in P-FAM	Estimation rule
1. Image analysis	$-$	$+$	Maximal time required to analyze single block
2. Statistics summation	$-$	$+$	$2 \cdot \{M + \lceil \log_2 P \rceil - \lceil \log_2 M \rceil\}$ if $P > M$, and $2 \cdot M$ if $P \leq M$, multiplied by the time required to add one entry in the model table to another
3. Model construction	$-$	$+$	$\lceil M/P \rceil$ times the time required to restore model for a single context
4. Model distribution	$-$	$+$	$\{M + \lceil \log_2 P \rceil - \lceil \log_2 M \rceil\}$ if $P > M$, and M if $P \leq M$, multiplied by the time required to copy single entry in the model table
5.1 Model initialization	$-$	$+$	Difference in time required for initialization of the model of the MQ-coder to particular initial model between initialization to zero-state
5.2 Image coding	$+$	$+$	Maximal time required to compress single block

Table III. Reference execution times. Here, P is the number of processors; b is the block dimension; N is the number of pixels in the block; T_X is the time required to execute Loop X, in ms.

	Block size		P-JBIG	P-FAM						
P	B	N	T_{TOTAL}	T_1	T_2	T_3	T_4	$T_{5.1}$	$T_{5.2}$	T_{TOTAL}
1	5000	25 000 000	4441	2714	0	0.211	0	0.005	4466	7180.2
4	2500	6250 000	1113.4	713.52	0.025	0.053	0.005	0.005	1113.6	1827.2
25	1000	1000 000	184.41	118.77	0.025	0.008	0.005	0.005	184.31	303.12
100	500	250 000	52.57	35.25	0.025	0.002	0.005	0.005	52.52	87.81
400	250	62 500	13.92	9.92	0.025	0.0006	0.005	0.005	13.92	23.87
2500	100	10 000	2.66	1.90	0.025	0.0002	0.005	0.005	2.70	4.63
10 000	50	2 500	0.81	0.60	0.025	0.0002	0.005	0.005	0.81	1.44

interleaving in the memory modules featuring realistic latencies. According to our experiments, the simulation cost is 3% for the *ParSum()* and 31% for the *Distribute()* routines. These are the only parts requiring tightly synchronous executions; other parts may use loose synchronization eliminating the need for cost analysis.

The execution times are summarized in Table III for a selected number of processors. The *speedup* is illustrated in Figure 8. The *speedup* is computed as a ratio between the sequential time (T_{SEQ}) and the parallel time (T_{PAR}) using P processors:

$$\mathit{speedup}(P) = T_{\text{SEQ}}/T_{\text{PAR}}$$

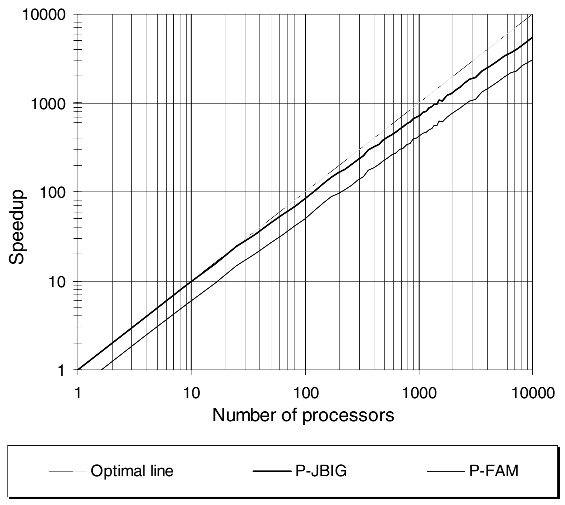

Figure 8. Speedup factor.

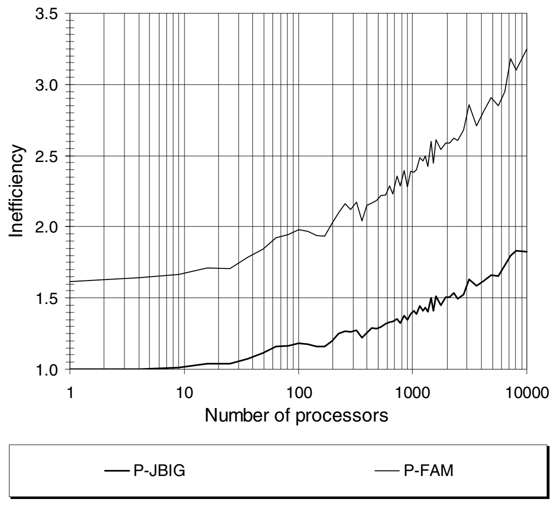

Figure 9. Inefficiency of the parallel algorithms.

Table IV. Reference code sizes. Here, P is the number of processors; b is the block dimension. Code size for the P-FAM method inclusive of model overhead (1024 bytes). Uncompressed image size is 3 125 013 bytes.

P	Block size $b \times b$	Code size, bytes	
		P-JBIG	P-FAM
1	5000×5000	295 611	296 436
4	2500×2500	296 632	296 997
25	1000×1000	300 441	298 712
100	500×500	300 634	298 104
400	250×250	305 451	299 411
2500	100×100	317 510	302 436
10 000	50×50	363 155	323 184

The optimal line in Figure 8 refers to the maximum speedup that can be obtained using P processors. The forward-adaptive variant (P-FAM) is about 1.6 times slower than the backward-adaptive variant (P-JBIG) because of the additional analysis stage.

The work *inefficiency* of the parallel algorithm is illustrated in Figure 9. It measures the factor of cost increase caused by using the parallel algorithm instead of the sequential, and can be calculated as

$$\mathit{inefficiency} = \mathit{cost}(P)/T_{\mathrm{SEQ}} = P/\mathit{speedup}(P)$$

The *cost* of executing an algorithm is defined as the overall workload of the algorithm. The cost of a sequential algorithm equals its execution time, whereas the cost of a parallel algorithm is calculated as the number of processors multiplied by the overall execution time:

$$\mathit{cost}(P) = P \cdot T_{\mathrm{PAR}}$$

The reference execution times are given in Table III. As can be seen, the inefficiency increases with the growth of the number of processors. For example, the costs of using P-JBIG and P-FAM with 10 000 processors are, respectively, 1.82 and 3.25 times higher than using JBIG.

Code sizes

Average code sizes for the reference numbers of parallel processors are given in Table IV, and the inefficiency of compression is illustrated in Figure 10. The *compression inefficiency* is measured as the factor of increase in the code size,

$$\mathit{compression\ inefficiency} = \mathit{code\ size}(P)/\mathit{code\ size}_{\mathrm{JBIG}}$$

where $\mathit{code\ size}(P)$ is the code size resultant of the parallel algorithm using P processors, and $\mathit{code\ size}_{\mathrm{JBIG}}$ is the code size resultant of the sequential algorithm, JBIG in particular. The experimental

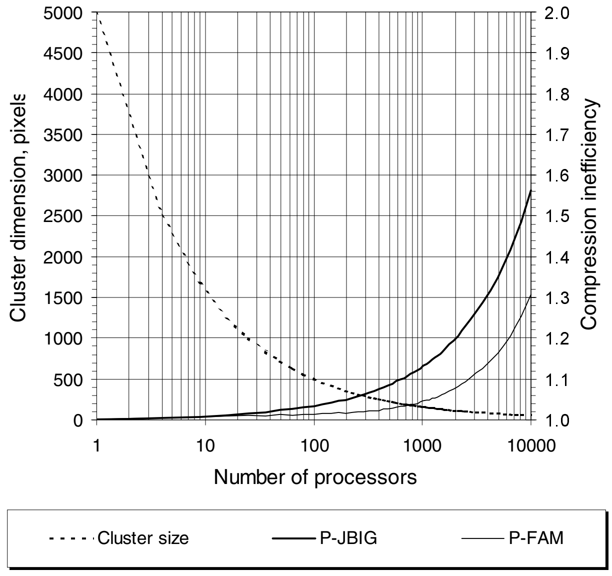

Figure 10. Compression inefficiency of the parallel implementation.

results show that the compression inefficiency of the P-JBIG remains within a 5% range when the number of processors is less than 170. The corresponding number of processors for the P-FAM method is 1024. For 10 000 processors, the code size increases by a factor of 1.56 in the case of P-JBIG, and 1.31 in the case of P-FAM.

CONCLUSIONS

In JBIG-based methods, the image is compressed sequentially using backward-adaptive context-based modeling and arithmetic coding. We have shown that this kind of sequential approach can be implemented efficiently also in parallel using block decomposition. The compressed blocks are stored into the same file and an index table is included to locate the starting points of the code blocks. Using the JBIG2 file format, the images are fully compatible with those that compressed sequentially. The use of parallelization, however, must be decided at the time of compression.

We have shown by experiments that the parallel implementation achieves rather good utilization of the processors. The inefficiency remains within the factor of two with any reasonable number of processors. A drawback of parallelization is the decrease of the compression performance when using a very large number of processors and, therefore, blocks. This drawback can be compensated by the use of forward-adaptive modeling. It slows down the compression but enables the use of smaller blocks and consequently, more processors with the same compression performance. Using the proposed P-FAM method, a 5000 $\times$ 5000 pixels map image can be divided into about 1000 blocks with only about a

5% increase in the code size. The corresponding speedup factor is about 400 if 1000 processors are employed.

As an additional benefit, the block decomposition provides direct access to the compressed image file. This property is useful when browsing very large images, or in real-time imaging applications and mobile environments, where the images must be stored in compressed form and only a fraction of the image needs to be viewed/accessed at a time.

ACKNOWLEDGEMENT

The project was funded by the grant 954/401/99 of the National Technology Agency of Finland (TEKES).

REFERENCES

1. JBIG. Progressive bi-level image compression. ISO/IEC International Standard 11544, 1993.
2. Final committee draft for ISO/IEC International Standard 14492, 1999. http://www.jpeg.org/public/jbigpt2.htm.
3. Howard PG, Kossentini F, Martins B, Forchammer S, Rucklidge WJ, Ono F. The emerging JBIG2 standard. *IEEE Transactions on Circuits and Systems for Video Technology* 1998; **8**(7):838–848.
4. Fortune S, Wyllie J. Parallelism in random access machines. *Proceedings of the 10th STOC*. ACM: New York, NY, 1977; 114–118.
5. Abolhassan F, Drefenstedt R, Keller J, Paul WJ, Scheerer D. On the physical design of PRAMs. *Computer Journal* 1993; **36**(8):756–762.
6. Keller J, Paul WJ, Scheerer D. *Proceedings of the WDAG '94, 8th International Workshop on Distributed Algorithms*, Terschelling, The Netherlands (*Lecture Notes in Computer Science*, vol. 857). Springer, 1994; 17–27.
7. Alverson R *et al.*. The Tera computer system. *Proceedings of the International Conference on Supercomputing*. ACM: New York, NY, 1990; 1–6.
8. Forsell M. Implementation of instruction-level and thread-level parallelism in computers. *Dissertations 2*, Department of Computer Science, University of Joensuu, Finland, 1997.
9. Keller J, Kessler C, Träff T. *Practical PRAM Programming*. Wiley: New York, NY, 2001.
10. Forsell M, Kumar S. Virtual distributed shared memory for network on chip. *Proceedings 19th IEEE NORCHIP Conference*, Kista, Sweden. IEEE, 2001; 192–197.
11. Ageenko E, Fränti P. Forward-adaptive method for context-based compression of large binary images. *Software—Practice and Experience* 1999; **29**(11):943–952.
12. Pennebaker WB, Mitchell JL, Langdon GG, Arps RB. An overview of the basic principles of the Q-coder adaptive binary arithmetic coder. *IBM Journal of Research and Development* 1988; **32**(6):717–726.
13. Pennebaker WB, Mitchell JL. Probability estimation for the Q-coder. *IBM Journal of Research and Development* 1988; **32**(6):737–759.
14. Henriques S, Ranganathan N. A parallel architecture for data compression. *Proceedings 2nd IEEE Symposium on Parallel and Distributed Processing*. IEEE, 1990; 260–266.
15. Howard P, Vitter J. Parallel lossless image compression using Huffman and arithmetic coding. *Proceedings of the Data Compression Conference DCC'92*, March 1992. IEEE, 1992; 299–308.
16. Horie H, Shirai H, Iizuka Y, Takahata M. Bi-level image high speed code conversion processor: ImPC2. *Proceedings 1998 IEEE Asia–Pacific Conference on Circuits and Systems*. November 1998. IEEE, 1998; 615–618.
17. Tarui M, Oshita M, Onoye T, Shirakawa I. High-speed implementation of JBIG arithmetic coder. *Proceedings IEEE TENCON 99*, September 1999. IEEE, 1999; 1291–1294.
18. Jiang J. Parallel design of Q-Coders for bilevel image compression. *Proceedings IEEE International Conference on Parallel and Distributed Systems*, December 1994. IEEE, 1994; 230–235.
19. Boliek M, Allen J, Schwartz E, Gormish M. Very high speed entropy coding. *Proceedings IEEE International Conference on Image Processing ICIP-94*, vol. 3. IEEE, 1994; 625–629.
20. Leighton TF. Introduction to parallel algorithms and architectures. *Arrays, Trees, Hypercubes*. Morgan Kaufmann: San Mateo, CA, 1992.
21. Brent R. The parallel evaluation of general arithmetic expressions. *Journal of the Association for Computing Machinery* 1974; **21**(2):201–206.