UNIVERSITY OF JOENSUU
DEPARTMENT OF COMPUTER SCIENCE
Report Series A

Context-based Compression of Binary Images in Parallel

Eugene Ageenko, Martti Forsell
and Pasi Fränti

Report A-2000-3

ACM I.4.2, I.5.3
UDK 519.712
ISSN 0789-7316
ISBN 951-708-948-1

Context-based Compression of Binary Images in Parallel

Eugene Ageenko, Martti Forsell, Pasi Fränti

Department of Computer Science, University of Joensuu, PB 111, FIN-80101 Joensuu, Finland.
Email: franti@cs.joensuu.fi.

Abstract

Binary images can be compressed efficiently using context-based statistical modeling and arithmetic coding. The problem of this kind of approach is that the process is fully sequential and therefore, additional computing power from parallel computers cannot be utilized. We attack this problem and show how the context-based compression can be implemented 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 how the construct, distribute and utilize the model in parallel, and study the effect on the compression performance and run time. We will show by experiments that the proposed approach achieves speedup that is proportional to the number of processors. The work efficiency of the extra processing exceeds 50% with any reasonable number of processors.

Keywords: image compression, context modeling, JBIG, parallel algorithms, EREW, PRAM.

1 Introduction

Binary images can be efficiently compressed using context-based statistical modeling and arithmetic coding. For example, the latest compression standards, JBIG and JBIG2 are both based on this approach [JBIG1, JBIG2]. The methods underlying these standards process the image pixel by pixel using backward adaptive modeling. The compression of a single pixel therefore depends on all the previous data. Moreover, the bit stream provided by arithmetic coding is unbreakable and the compression is therefore completely sequential in its nature.

Images in question are binary images such as text documents and fax messages, or images composed of a limited number of binary layers. This kind of images have typically only a few color tones but high spatial resolution to represent fine details such as text and graphics objects, shape information, and elevation lines of maps. The size of the images vary from about 1000×2000 (fax images) to 5000×5000 (map images). For example, the map images can be compressed usually by a factor about 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 few bit-level operations of the arithmetic coder without any multiplications. However, there can be several millions of pixels to be compressed and the overall compression time can vary from several seconds to several minutes. This can degrade the usability of the compression in on-line applications. For example, let us consider the decompression of a 5000×5000 image taking 300 kilobytes of disc space. This amount of information can be read within a fraction of a second but to decompress, it takes several seconds even on modern Pentium-III based workstations.

The research toward parallel computers is still far from being complete but it is inevitable that parallel computers will eventually take the place of the current sequential computers in everyday life. For example, suppose that we have a workstation with 10 processors instead of the 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 if the image were decompressed within a second or two, instead of 10 or 20 seconds if there are computing potential to do so.

Parallel random access machine (PRAM), first proposed by Fortune and Wyllie in 1978 [Fortune78], is a general model for parallel computation. While building of shared memory has remained unfeasible, there has been a considerable effort to simulate an ideal shared memory machine like a PRAM with a physically distributed memory machine, which consists of processors and memory modules connected to each others through a communication network [Schwartz80, Ranade87, Abolhassan93, Forsell97]. A few experimental and commercially available PRAM-based computers have recently appeared. Among them are *Saarbrücken Parallel Random Access Machine* (SB-PRAM) [Abolhassan93] based on Fluent Abstract Machine [Ranade87], Berkley RISC I style multithreaded processor [Keller94] and *Cray MTA* supercomputer from Cray Inc (former Tera Computer Corporation) [Alverson90].

The problem of running backward adaptive compression method under a parallel model, such as PRAM, is that the compression is fully sequential, and the extra computing power cannot be therefore utilized. In this paper, we attack this problem and present a method for context-based compression of binary images in parallel. Our approach is to divide the compression task into several smaller tasks by dividing the image into $b \times b$ non-overlapping rectangular blocks. These blocks are processed independently on parallel processors.

The compressed blocks are stored in the same file in a sequential manner assuming that the file I/O is not a bottleneck. Index table must be stored in the header of the file to locate the starting points of the code blocks. In this way, decompression can also be performed in parallel. The JBIG2 file structure supports this kind of file organization where the image is divided into separate code blocks, and each block is stored in its own file segment. Each processor can then decompress from one to several image blocks depending on the number of processors available. In the following, we assume for simplicity that the number of blocks equals to the number of processors.

We will experimentally show that the proposed approach is efficient. It achieves speedup that is proportional to the number of processors. The inefficiency of parallel implementation remains within the factor of 2 when using any reasonable number of processors. A minor 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, in which the blocks share the same statistical model optimized for the entire image. We will give solutions how the construct, distribute and utilize the model in parallel, and study the effect on the compression performance and run time.

The rest of the paper is organized as follows. Context-based compression is briefly recalled in Section 2. The image compression using parallel random access machine is outlined in Section 3. The extension to forward-adaptive modeling and the implementation details are studied in Section 4. The experiments are given in Section 5, and the conclusions are finally drawn in Section 6.

2 Context-based compression

The image is processed pixel by pixel in raster-scan order starting from the top leftmost pixel of the image as shown in Figure 1. In context-based modeling, the probability estimation of the black and white pixels is conditioned on 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. The estimation starts from scratch and adapts the model to the input data.

The approach is adopted in the international standards by ITU-T: the JBIG (*Joint Bi-level Image Group*) [JBIG1], and the emerging standard [JBIG2, Howard98]. The 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 will remain similar to JBIG with the difference that a newer version of the coder is used, namely the MQ-coder.

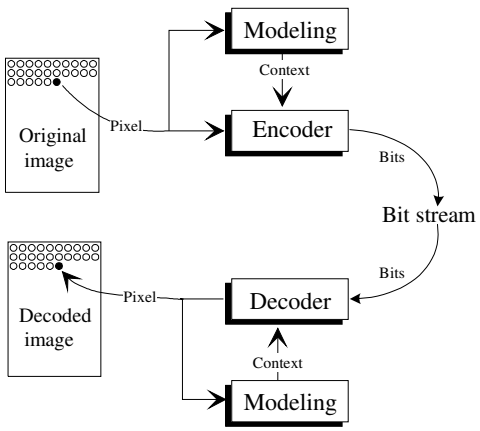


Figure 1. Block diagram of backward-adaptive context-based compression.

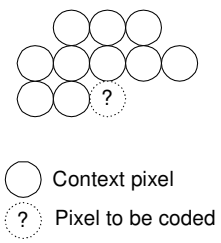


Figure 2. Example of a template of 10 pixels resulting in 1024 contexts.

The MQ-coder is an approximate implementation of arithmetic coding tailored for binary data. It's pre-ancestor, Q-coder, has similar working principles and is introduced in [Pennebaker88a]. The coder sub-optimality is compensated by the sophisticated automaton-based probability estimation, providing fast adaptation to the source data.

The probability estimation in the MQ-coder is derived from the arithmetic coder renormalization and is based on the Bayesian estimation concept [Pennebaker88b]. Instead of maintaining pixel counts, the estimation process is implemented as a state automaton consisting of 94 states (QM-coder had 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. Each context has its own pointer to the automaton, where one bit indicates the color of *more-probable symbol* (MPS). The automaton has mirror symmetry about the change in the sense of MPS color, and we therefore consider only 47 states, see Figure 3.

The adaptation process starts from blank model represented by *zero-state*. 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 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.

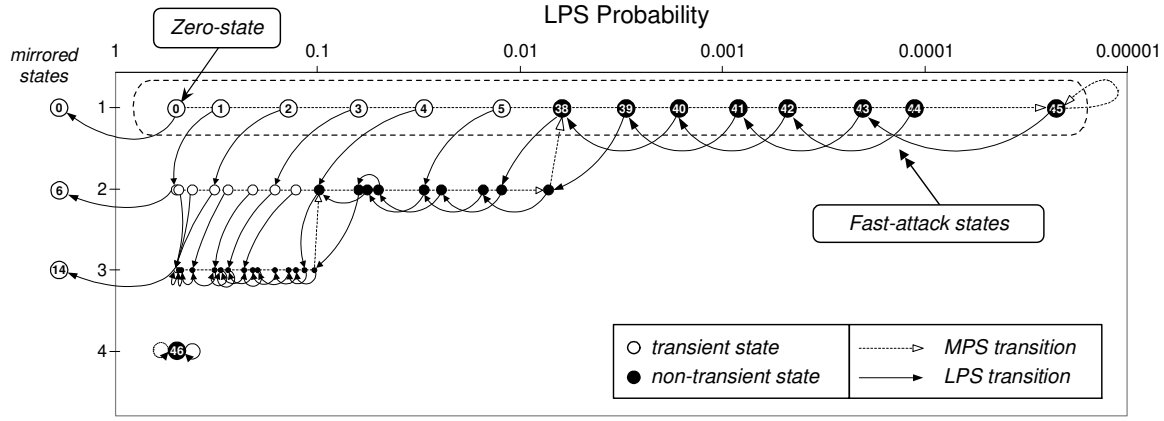


Figure 3. Spatial organization of the state automaton in MQ-coder and the transition sketch for the fast-attack states. Because of the mirror symmetry regarding the change in sense of LPS and MPS, only half of the states are depicted.

3 Parallel compression

One way to uncover the strength of efficient parallel processing is to concentrate on the computation and to ignore other practical considerations of parallel processing such as communication between processors. One can simply assume that the communication does not cost anything. A computational model that embodies this assumption is the *parallel random access machine* (PRAM), first proposed by Fortune and Wyllie in 1978 [Fortune78] as a general model for parallel computation. Although the PRAM model is somewhat idealistic, a substantial number of efficient PRAM simulations have been designed that allow efficient implementation on real parallel architectures. Moreover, the PRAM may be viewed as a virtual parallel machine and could serve as a design model for the development of very fast computational algorithms.

3.1 EREW PRAM model

A PRAM 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 run 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 level of shared memory access [Leighton91]. In this paper, we refer to just one of them, to *EREW* (Exclusive Read Exclusive Write). 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 *FOR .. PARDO* -statement, which executes the commands inside the loop synchronously in parallel. Using this command, a simple loop that consists of n iteration and does not access 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 *ParSum*() -routine for computing the sum of n numbers. Because of the limitations of EREW, it can be performed in $O(\log n)$ time. The third tool is the datum broadcast *Distribute*(), which requires $O(\log n)$ time to distribute single datum to n processors.

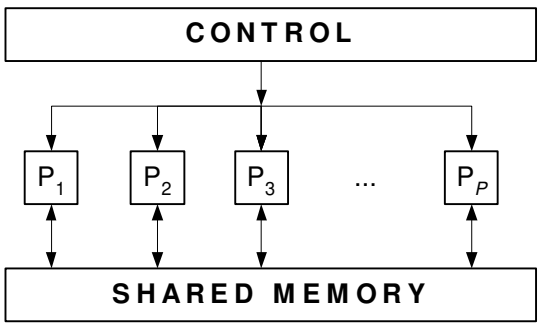


Figure 4. The PRAM model. $P_1, \ldots, P_P$ are processors.

3.2 Parallel Implementation

The proposed parallel implementation (referred here as P-JBIG) is based on decomposition of the image into B equally sized blocks consisting of $b \times b$ pixels each. Each block is then compressed separately by a processor dedicated to the block. This provides a simple division of the problem into parallel sub-problems. Furthermore, 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, see Figure 5.

For simplicity, we will assume that the number of blocks B equals to 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 smaller than the number of blocks then more than one blocks is processed with single processor. If the number of processors is greater than the number of blocks then we de-allocate additional processors for other purposes.

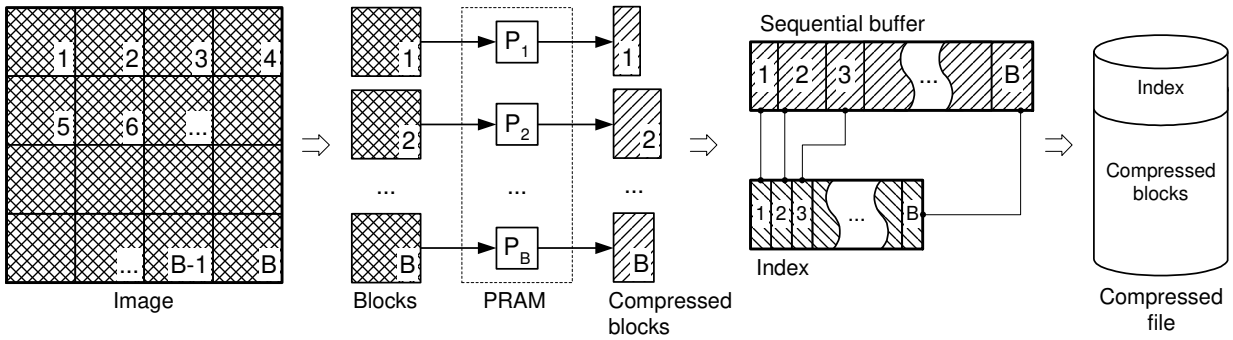


Figure 5. Parallelization using block decomposition and indexing.

4 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, and this increases the learning cost. Another problem related to the compression of small blocks is the inefficiency near block boundaries because the pixels from the neighboring blocks cannot be utilized. Previous studies indicate that the compression inefficiency remains tolerable if the block size is kept 256×256 or higher [Ageenko99].

In order to compensate the compression inefficiency, we consider a forward-adaptive variant of the parallel compression. The proposed method is based on the ideas in [Ageenko99], and is referred here as the P-FAM (*Parallel forward-adaptive model*). The method makes two passes over the image. In the first pass, statistics are collected from the image and models are optimized on the basis of the statistics. In the second pass, the blocks are compressed using the constructed models. The models are stored in the compressed file. The pseudo-code of the P-FAM is outlined in Figure 6, and will be detailed in the following subsections.

```
FOR p := 1 TO P PARDO                                   {LOOP 1: Analyse image}
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;

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

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

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

FOR p := 1 TO P PARDO                                   {LOOP 5: Compress image}
BEGIN
    FOR m := 0 TO M DO                                  {LOOP 5.1: Initialize the models}
    BEGIN
        InitializeModels (m, state[m, p]);
    END;
    FOR (each pixel x of block p) DO                    {LOOP 5.2: Compress the blocks}
    BEGIN
        c := DetermineContext(x);EncodePixel(x, c);
    END;
END;
```

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

4.1 Modeling

In the first phase, the input image is analyzed and the models are constructed. Statistics are first collected from the image by summing up the number of white and black pixels. Each processor analyses a single block using its own counters (Loop 1). After the blocks have been processed, the statistics are summed in parallel into the first column of the counter tables (Loop 2). The statistics are then translated to the nearest fast-attack states in the state automaton of the MQ-coder (Loop 3). Finally, the resulting models (states) are then distributed to every processor (Loop 4). The distribution of the models is necessary due to the access limitations of the EREW PRAM model.

4.2 Coding and decoding

In the coding phase, the blocks are compressed essentially in the same way as in Section 3. The only difference is that the models are initialized using the states obtained in the analysis phase (Loop 5.1). The initialization 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.

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.

4.3 Run time analysis

In the following analysis, we ignore the I/O and cost of EREW PRAM simulation. The asymptotic estimations of the complexity of the different loops of P-FAM are summarized in Table 1. 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 this must be performed by every processor. The encoding step (Loop 5.2) requires $O(N/P)$ time similarly 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 i-th iteration, P/i columns of the length M are summed in parallel using 2^i processors for each column. Therefore, the complexity totals to:

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

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

$$O(M/2) + O(M/4) + \ldots + O(M/M) + O(1) + \ldots + O(1) = O(M) + O(\log_2 P) - O(\log_2 M) = O(M + \log_2 \{P/M\}).$$

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

Table 1. The computational complexity of the P-FAM algorithm. $N \equiv P$ is assumed.

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

The total time complexity of the sequential and parallel implementations are $O(N)$ and $O(N/P) + O(M+\log_2 \max\{1, P/M\})$ respectively. For large image sizes and relatively low number of blocks (processors), the parallel implementation is nearly P times faster than the sequential. The relative speedup of the parallel implementation regresses when the number of processors increases. This is because the term $\log_2 (P/M)$ starts to dominate the time complexity for larger values of P. This has only 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 $P_{MAX} = 10\,000$.

5 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 PC based on 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 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 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.

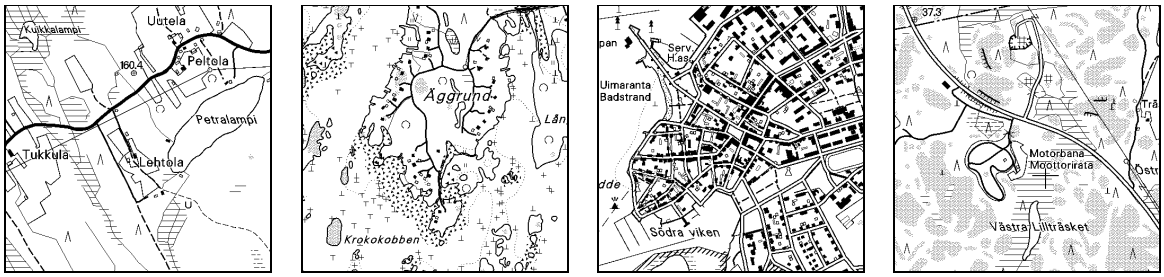


Figure 7. Sample fragments of the test images. All fragments of the size 500×500 pixels.
Reprinted with the permission of National Land Survey of Finland.

5.1 Run time

The execution time simulation is performed using sequential implementation of the algorithm. The actions taken to simulate various stages of the algorithm are illustrated in Table 2. Function *clock* (ANSI C) is used for measuring the time intervals. This function returns the number of clock ticks of the elapsed processor time since the start of a process. Because of a multi-tasking environment, the processes running on most modern operating systems, including Windows 2000, usually receive execution focus in time intervals with the length of 10 ms. Therefore, the accuracy of time computation equals ±1 interval = 20 ms. To alleviate this limitation, we have repeated the processes from 100 to 100 000 times and have measured the total execution time. The compression of some block may take longer that the compression of another one. This 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 to the maximum time required to compress a single block.

Table 2. 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 $P \le M$, 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 $P \le M$, 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

Since the operations of decompression are similar to that of the compression, and because of the fact that coding/decoding process is symmetrical, we only present the compression times. The execution times are summarized in Table 3 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_{JBIG}) and the parallel time T_P using P processors:

$Speedup\ (P) = T_{JBIG} / T_P$,

The optimal line in Figure 8 refers to the maximum speedup that can be obtained using P processors. The P-FAM method is about 1.6 times slower than the P-JBIG because of the additional analysis stage. The *inefficiency* of the parallel algorithm measures the factor of cost increase caused by using the parallel algorithm instead of sequential [Kronsjö96]. The *cost* of using a parallel algorithm is defined as product of its execution time T_P and the number of processors P :

$Cost\ (P) = P \cdot T_P$.

The cost of a sequential algorithm equals to its run time. Thus, the inefficiency of a parallel algorithm is computed as:

$Inefficiency = Cost\ (P) / T_{JBIG} = P / Speedup\ (P)$.

The inefficiency of the parallel implementations is illustrated in Figure 9. As we can see, 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.

Table 3. Reference execution times. Here, P is the number of processors; b is the block dimension; n is 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	6 250 000	1113.4	713.52	0.024	0.053	0.004	0.005	1113.6	1827.2
25	1000	1 000 000	184.41	118.77	0.024	0.008	0.004	0.005	184.31	303.12
100	500	250 000	52.57	35.25	0.024	0.002	0.004	0.005	52.52	87.81
400	250	62 500	13.92	9.92	0.024	0.0006	0.004	0.005	13.92	23.87
2 500	100	10 000	2.66	1.90	0.024	0.0002	0.004	0.005	2.70	4.63
10 000	50	2 500	0.81	0.60	0.024	0.0002	0.004	0.005	0.81	1.44

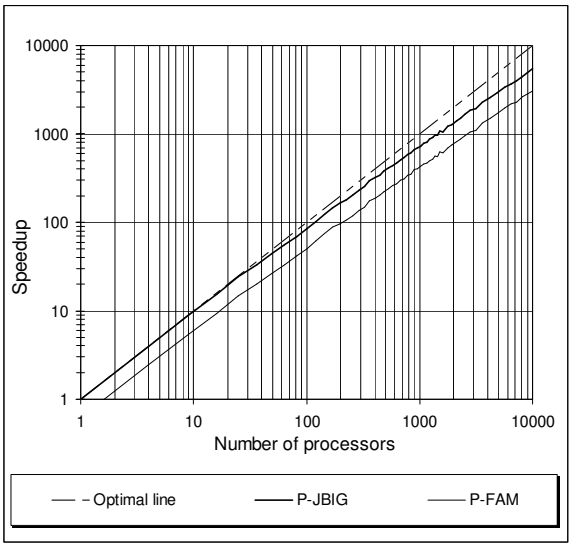


Figure 8. Speedup factor.

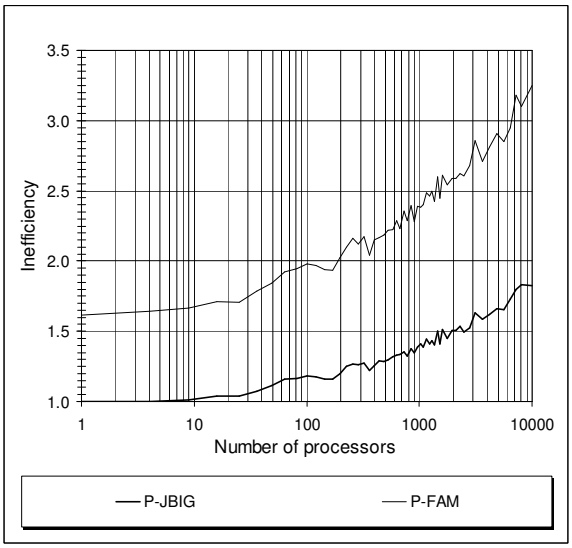


Figure 9. Work efficiency of the parallel algorithms.

5.2 Code sizes

Average code sizes for the reference numbers of parallel processors are given in Table 4, and the inefficiency of compression is illustrated in Figure 10. The *compression inefficiency* is measured as the factor of increase in the codesize:

$Compression\ inefficiency = Code\ size\ (P) / Code\ size_{JBIG}$,

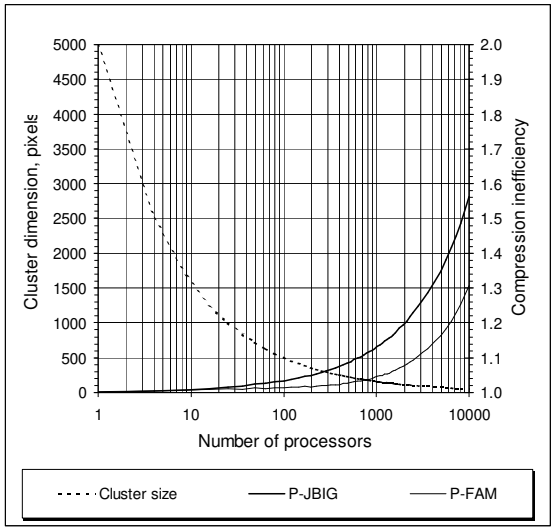
Where $Code\ size\ (P)$ is the code size resultant of parallel algorithm using P processors.

The experimental results show that the compression inefficiency of the P-JBIG remains within 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 factor of 1.56 in the case of P-JBIG, and 1.31 in the case of P-FAM.

Table 4. Reference code sizes. Here, *P* is the number of processors; *b* is block dimension. Code size for P-FAM method inclusive model overhead (1024 bytes). Uncompressed image size is 3 125 013 bytes.

	Block size	Code size, bytes	
P	*b*×*b*	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
2 500	100×100	317 510	302 436
10 000	50×50	363 155	323 184

Figure 10. Compression inefficiency of the parallel implementation.



6 Conclusions

In JBIG-based methods, images are 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 JBIG2 file format, the images are fully compatible with those of 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 2 with any reasonable number of processors.

Additional benefit of the block decomposition is that we have direct access to the compressed image file. This property can be useful when browsing very large images, or in real-time imaging applications of mobile environments where the images must be stored in compressed form and only a fraction of the images can be viewed at a time. The side effect of the parallelization is that the compression performance decreases when using very large number of processor. This can be compensated by the use of forward-adaptive modeling. It slows down the compression of a single block but allows the use of more dense block decomposition and hence, more processors with the same compression performance. With standard A4 size images we can be divide the image into about 400 blocks with only about 5 % increase in the code size. The corresponding speedup factor is about 200 if 400 processors are employed. Using proposed P-FAM method, a standard map image can be divided into about 1000 blocks with only about 5 % increase in the code size. The corresponding speedup factor is about 400 if 1000 processors are employed.

7 References

[JBIG1]

ISO/IEC International Standard 11544 (1993), *ISO/IEC JTC 1/SC 29/WG 9*; also ITU-T Recommendation T.82. Progressive Bi-level Image Compression.

[JBIG2]

Final committee draft for ISO/IEC International Standard 14492 (1999), *ISO/IEC JTC 1/SC 29/WG 9*. http://www.jpeg.org/public/jbigpt2.htm

[Howard98]

Howard P.G., Kossentini F., Martins B., Forchammer S., Rucklidge W. J., Ono F. (1998), The emerging JBIG2 standard. *IEEE Trans. Circuits, Systems for Video Technology* 8 (7): 838-848.

[Ageenko99]

E. Ageenko, P. Fränti (1999), Forward-Adaptive Method for Context-Based Compression of Large Binary Images, *Software - Practice and Experience* 29 (11): 943-952.

[Pennebaker88a]

W.B., Mitchell J.L., Langdon G.G., Arps, R.B (1988), An overview of the basic principles of the Q-coder adaptive binary arithmetic coder. *IBM Journal of Research, Development* 32 (6): 717-726.

[Pennebaker88b]

Pennebaker W.B., Mitchell J.L. (1988) Probability estimation for the Q-coder. *IBM Journal of Research, Development* 32(6): 737-759.

[Fortune78]

S. Fortune and J. Wyllie (1978), Parallelism in Random Access Machines, *of 10th ACM STOC* (Association for Computing Machinery, New York): 114-118.

[Leighton91]

T. F. Leighton (1992), Introduction to Parallel Algorithms and Architectures: in Arrays, Trees, Hypercubes, Morgan Kaufmann, San Mateo.

[Schwarz80]

J. T. Schwarz (1980), Ultracomputers, *ACM Transactions on Programming Languages and Systems* 2 (4): 484-521.

[Ranade87]

A. G. Ranade, S. N. Bhatt, S. L. Johnson, (1987) The Fluent Abstract Machine, *Technical Report Series BA87-3* (Thinking Machines Corporation, Bedford).

[Abolhas93]

F. Abolhassan, R. Drefenstedt, J. Keller, W. J. Paul and D. Scheerer (1993), On the Physical Design of PRAMs, *Computer Journal* 36 (8): 756-762.

[Forsell97]

M. Forsell (1997), Implementation of Instruction-Level and Thread-Level Parallelism in Computers, *Dissertations 2* (Department of Computer Science, University of Joensuu, Joensuu).

[Keller94]

J. Keller, W. Paul, D. Scheerer (1994), Realization of PRAMs: Processor Design, *Proceedings of WDAG '94, 8th International Workshop on distributed Algorithms* (Terschelling, The Netherlands): 17-27.

[Alverson 90]

R. Alverson, D. Callahan, D. Cummings, B. Kolblenz, A. Porterfield, B. Smith, The Tera Computer System (1990), *Proc. of the International Conference on Supercomputing* (Association for Computing Machinery, New York): 1-6.

[Kronsjö96]

L.I. Kronsjö (1996), Pram Models, in *Parallel and Distributed Computing Handbook*, (ed. A. Zomaya), McGraw-Hill, pp. 163-191.