Theoretical and Applied Informatics
ISSN 1896–5334
Vol. 19 (2007), no. 2
pp. 103–115

Parallelization of an adaptive compression algorithm using the reduced model update frequency method

Roman Starosolski [a]

[a]Institute of Computer Science,
Silesian University of Technology
Akademicka 16, 44-100 Gliwice, Poland
roman.starosolski@polsl.pl

Received 22 September 2007. Revised 10 October 2007. Accepted 23 October 2007.

Abstract: Modeling and coding is the most complex part of many adaptive compression algorithms; it is also an inherently serial process. The method of reduced model update frequency is a modification of the typical adaptive scheme, which was originally employed in order to improve the speed of modeling at the cost of a negligible worsening of the modeling quality. In this paper, we notice that the above method permits to parallelize the compression process. We find that for the Itanium 2 processor the speed of coding and modeling in a SFALIC image compression algorithm may be improved by about 50% through exploiting the fine-grained parallelism, also the medium-grained parallelism may be exploited in a significantly larger extent.

Keywords: Parallel processing; Fine- and medium-grained parallelism; Adaptive modelling; Data compression; Image compression

1. Introduction

Designing compression algorithms, we aim at encoding the given input sequence of symbols using the smallest number of bits. In order to encode the symbol s optimally, we should use -$\log_2$(prob(s)) bits, where prob(s) is the probability of the symbol s occurrence [12]. Employing an arithmetic coder [10] we may get arbitrarily close to the above optimum, however practical implementations of the arithmetic coding are relatively slow. We may also use prefix codes, such as Huffman codes [7]. In this case we encode each symbol from a sequence with a binary codeword of integer length. Prefix codes may be inefficient if the probabilities are high, but in practice are much faster.

Huffman codes may be used for any probability distribution; there are also prefix codes optimal for certain probability distributions, like Golomb [5] or Golomb-Rice [11] families optimal for exponential distribution. Knowing the type of the distribution permits faster coding and does not require estimating of the whole unknown distribution – instead we estimate the parameter of a known distribution. The probability distribution is estimated by the data model. If the model estimates conditional probabilities, i.e., if the specific symbol's context is considered in determining symbol's probability, then it is the context model, otherwise the model is memoryless. In the adaptive modeling, the distribution for specific symbol is estimated based on statistics gathered from the already processed symbols. This way, as opposed to two-pass schemes, we do not have to transmit explicitly the data model since the decompression algorithm is able to reconstruct the model on-line from already decoded symbols. After encoding of the symbol we update the data model (Fig. 1). The coding and modeling process is inherently serial; before estimating the probability distribution for the next symbol, and before encoding the next symbol, we have to update the model with information on the previous symbol.

```
loop
  read symbol s
  estimate coding parameter r for symbol s
  encode symbol s using r
  update model with symbol s
endloop
```

Fig. 1. Adaptive modeling and coding

In the SFALIC lossless image compression algorithm [14] a method of reduced model update frequency (RUF) was employed. The motivation for introducing the RUF method is the observation of typical image characteristics that change gradually for almost all the image area or even are invariable. In order to adapt to gradual changes, we may sample the image, i.e., update the data model, less frequently than each time the pixel gets coded. Instead, we update the model after coding of selected pixels only. We could simply pick every *i*-th symbol to update the model, but such a constant period could interfere with the image structure. Therefore each time, after updating the model with some symbol, we select randomly a number of symbols to skip before the next update of the model (Fig. 2). In order to permit the decoder to select the same number we use pseudo-random number generator. By selecting the range of the pseudo-random numbers we may change the model update frequency, i.e., the probability of updating the model after coding a symbol. This way we control the speed of adapting the model and the speed of the compression process. At the beginning of the compression, the data model should adapt to the image data characteristics as quickly as possible, so we start the compression using all symbols for updating the model (i.e., we use the update

frequency of 100%) and then we gradually decrease the model update frequency, until it reaches some minimal value. The RUF method allowed significant improvements of compression speed at the expense of worsening the compression ratio insignificantly. Using the update frequency of 3.08% (i.e., the default update frequency of the SFALIC algorithm) we got 3 times greater compression speed and 0.5% worse compression ratio compared to the update frequency of 100%.

```
delay := random(range)
loop
  read symbol s
  estimate coding parameter r for symbol s
  encode symbol s using r
  if delay = 0 then
    update model with symbol s
    delay := random(range)
  else
    delay := delay-1
  endif
endloop
```

Fig. 2. The reduced update frequency (RUF) method

Many lossless image compression algorithms, including SFALIC, are predictive. In a predictive algorithm we do not encode pixels explicitly, instead we use a predictor function to guess the pixel intensities and then we calculate prediction errors, i.e., differences between actual and predicted pixel intensities. Next, we encode the sequence of symbols, which are prediction errors. To calculate the predictor for a specific pixel we usually use intensities of small number of already processed pixels neighboring it. For typical grayscale continuous-tone images, the pixel intensity distribution is close to uniform. Prediction error distribution is close to Laplacian, i.e., symmetrically exponential [3], making prediction errors easier and faster to compress.

The detailed description of the SFALIC algorithm exceeds the scope of this paper, hence we just briefly characterize it here and refer the reader to [14] for details. It is a predictive algorithm; for prediction we use a couple of already-processed neighbors of a specific pixel. A sequence of prediction errors is encoded using a modified Golomb-Rice family of limited codeword length. A context data model, based on a model of the FELICS algorithm [6], is used to adaptively estimate the coding parameter.

The RUF method was introduced in order to improve the compression speed by means of reducing the time spent, by the compression algorithm, on updating the data model. However, we found that this method permits to parallelize the compression process. In the fine-grained parallelism or instruction-level parallelism single CPU performs in parallel several operations of a single process, which is possible since contemporary

CPUs contain multiple execution units capable of parallel processing. By an analogy to multiprocessor systems (for introduction and classification of parallel processing architectures see e.g. [4]), we may recognize two types of fine-grained parallelism: MIMD and SIMD. In the Multiple-Instruction-Multiple-Data (MIMD) fine-grained parallelism several execution units of a single CPU operate in parallel – each of them executes its own instruction. In the Single-Instruction-Multiple-Data (SIMD) parallelism we execute in parallel the same instruction for multiple arguments. Contemporary CPUs contain special SIMD execution units designed exclusively for operating on multiple arguments, actually packed into special, long registers. Obvious candidates for parallelization are inner loops. In the case of the SIMD processing several iterations of the loop body are performed in parallel, this technique is called loop vectorization. We may employ SIMD processing provided that there are no inter-iteration dependencies and that the same sequence of operations is executed for each iteration, e.g. that there are no operations performed conditionally. In the MIMD processing the parallelization is possible even when both the above conditions are not met – since for consecutive iterations we may perform in parallel separate parts of the loop body (these parts have to be free of inter-iteration dependencies; the technique is called software pipelining [2]). In the medium-grained parallelism we execute multiple threads using multiprocessor system.

The reminder of this paper is organized as follows. The idea of parallelizing the adaptive image compression algorithm is introduced in section 2. In section 3 we analyze experimentally effects of the fine-grained parallelization of coding and modeling in the SFALIC algorithm and then we estimate effects of medium-grained parallelization of the whole compression process. Section 4 summarizes the research.

2. Idea of parallelization

In the RUF method the model is updated after coding of some symbols only (recall Fig. 2); we will call the sequence of symbols encoded between two consecutive model updates the block of symbols. Since for a block of symbols the model is constant, operations performed using the data model for symbols from the block may run in parallel. Therefore, in parallel we may find coding parameters using the data model as well as generate codewords. Provided that the length of the block of symbols is sufficient, parallel processing of pixels within the block should result in a speedup of coding and modeling. The parallelizable variant of the RUF method (RUF-P) is presented on the Fig. 3. The only operation that still has to be performed sequentially for each of the block symbols is outputting the variable length binary codewords. Storing the generated codewords might be a problem, because the codeword lengths are variable. In the SFALIC algorithm, however, the codeword length is limited to 31 bits, so the codewords may be stored as pairs of integers <codeword_length, codeword_bits>.

For RUF-P method in the SFALIC algorithm the MIMD processing may be effective. The steps of coding parameter determination and codeword generation require just several simple integer operations (table lookup, AND, OR, add, subtract, shift) feasible for typical CPUs' execution units, including SIMD units. Unfortunately they contain alternative paths of symbol processing selected dynamically based on the symbol value which makes the body of the `parfor` loop in Fig. 3 not implementable as a sequence of SIMD operations (although some operations inside this loop might be implemented as SIMD operations). The fine-grained MIMD parallelization of this loop may be effective, provided that the CPU contains sufficient resources (non-SIMD execution units and general-purpose integer registers).

```
delay := random(range)
loop
  read symbols s0 … sdelay
  parfor i:=0 to delay
    estimate coding parameter ri for symbol si
    generate codeword cwi of symbol si using ri
  endparfor
  output codewords cw0 ... cwdelay
  update model with symbol sdelay
  delay := random(range)
endloop
```

Fig. 3. The idea of fine-grained parallelization using the RUF method (RUF-P)

The RUF method permits to parallelize operations performed using the data model. The outputting of the already-generated variable length codewords cannot be parallelized itself; it must be performed sequentially for all the symbols we compress. Fortunately it may be performed in parallel to updating of the data model or, provided that we use two arrays of generated codewords, in parallel to determination of coding parameters and generation of codewords for the next group of symbols. So in this part of the compression process may exploit the medium-grained parallelism.

As mentioned above, the model may be updated while outputting the codewords. Actually almost all of the model update operations may also be performed in parallel to determination of coding parameters and generation of codewords. To start updating the model we need to know the symbol, after encoding of which, the model should be updated – this information is available as soon as the number of symbols to be encoded before the next model update is known and the symbols are read. The data model of the adaptive compression algorithm stores data on characteristics of the already processed symbols, which is needed to select a coding parameter. In the context model we store such data for each context separately. Knowing a symbol, along with its context, that

after encoding of the block of symbols should be used to update the model, we can calculate new data describing characteristics for the context in parallel to encoding of the block. After encoding, the context data in the model may be updated by just overwriting context data with updated information. Alternatively, for memoryless model, we may have two models and switch between them. Yet another possibility is natural to the RUF method. We use the model to find the coding parameter, but since we do not update the model each time we retrieve the parameter, it's better to find the parameter after updating the model and store it, than to find it each time the parameter is needed. In the SFALIC's context data model, an additional structure stores coding parameters selected for contexts (single integer per context). Only the updating of this additional structure, i.e., single assignment operation per model update, has to be performed after determination of parameters for block of symbols.

3. Experimental results and discussion

For experiments we used the unmodified implementation of the SFALIC algorithm, version 04, as well as its variant modified as in Fig. 3. Actual differences between these two implementations were smaller, than it appears from Figures 2 and 3. Unmodified implementation (RUF) is optimized and also processes symbols in blocks (Fig. 4). On the other hand the RUF-P variant is implemented using the `for` keyword, not the `parfor` (recall RUF-P variant in Fig. 3), so it's the compiler's responsibility to parallelize this loop exploiting CPU's resources. Among other things, in order not to favor the RUF-P variant, we relied solely on compiler abilities to recognize and take advantage of the parallelizable code, i.e., we did not use explicit parallel programming tools like e.g. MPI. Both implementations process images in rows: each row of image pixels is read from the input file, then the prediction for the whole row is performed and a resulting row of residuum symbols is stored in a memory buffer. Then modeling and coding takes place resulting in row of residuum symbols being encoded to another memory buffer, which is written to the output file afterwards.

Most of experiments were performed using an Itanium 2 processor because of its large number of resources (execution units and registers) [9]. This CPU contains 9 integer arithmetic execution units (6 of them being general-purpose ones), 10 multimedia execution units (capable of SIMD integer processing, 6 of them general-purpose), and 128 general-purpose 64-bit integer registers. For experiments we employed a computer equipped with two Itanium 2 processors (1.4 GHz, 256 kB L2 cache, 3 MB L3 cache) running Linux (kernel 2.4). However, in order to evaluate the fine-grained parallelism of a single CPU, the application executables were compiled as single-threaded. We used Intel optimizing C++ Compiler 9.0. The implementations used permit to skip certain stages of the compression process (writing output file, modeling and coding, etc.) so we

were able to measure precisely real execution time of selected stages of the compression process by subtracting results obtained for compression with certain stage skipped or left. For experiments we used for a set of 12 big natural continuous-tone grayscale images, of depths from 8 to 16 bits per pixel and approximate size of 4 millions pixels; images are from a set described in [13] (group "*big*"). As mentioned in the introduction, the update frequency is variable; compression process starts with updating the model after encoding of each symbol, and then the frequency is gradually decreased until it reaches specified destination frequency value. In order to obtain results for a certain constant update frequency only, we subtracted results obtained for upper half of the image from the results for whole image. We also checked that performing measurements just for whole image would change results by about a percent or less (which was expected since modeling frequency stops decreasing after algorithm processes first 12 thousands of pixels – 0.3% of the whole image). The execution time was measured by running the executables several times. The time of the first run was ignored and the collective time of other runs (executed for at least two seconds) was measured and then averaged. The time measured was a sum of user and system time, as reported by the operating system after application execution. The results are expressed in average number of ticks (CPU clock cycles) per image pixel, or in percents relative to the time of whole compression process.

```
delay := random(range)
loop
  read symbols s0 … sdelay
  for i:=0 to delay
    estimate coding parameter r for symbol si
    generate codeword cw of symbol si using r
    output codeword cw
  endfor
  update model with symbol sdelay
  delay := random(range)
endloop
```

Fig. 4. Actual implementation of the RUF method in the unmodified SFALIC implementation

We have compared speeds for the default SFALIC model update frequency (3.08%). The parallelized version RUF-P requires on average 22.3 ticks per pixel to find the coding parameter, generate the codeword, output the codeword and for some pixels only to update the model. The unmodified version requires 32.9 ticks, so the speedup is 1.47. The coding and modeling speed increase by about 50% may be practically useful, justifying the effort to alter the implementation sources, especially if we notice that it is the speedup of the whole modeling and coding process, not of just the parallelizable part of it.

On the Fig. 5 we report speedups for various update frequencies available in the implementations. As could be expected, for update frequencies greater than a certain threshold (22.2%) the RUF-P variant is slower than the unmodified version and the RUF-P variant speedup increases as the update frequency decreases. Apparently, for update frequencies greater than 22.2% (which implies very small sizes of blocks of symbols that are processed in parallel – on average 2.5 symbols or less) the cost of parallelization (e.g. of storing generated codewords in a memory buffer and of retrieving them) is greater than the savings due to parallel processing of symbols within the block. Therefore the speedup would be greater if we used update frequency smaller then the default one, however, the performance of the lossless image compression algorithm is not judged based solely on the compression speed; the usual primary criterion is the compression ratio. For the whole set from which test images were taken, using the default model update frequency worsened the average compression ratio by 0.5% compared to updating the data model each time a pixel gets coded; below the default update frequency ratios start to worsen much faster so using smaller frequencies for this algorithm may in practice be not justified.

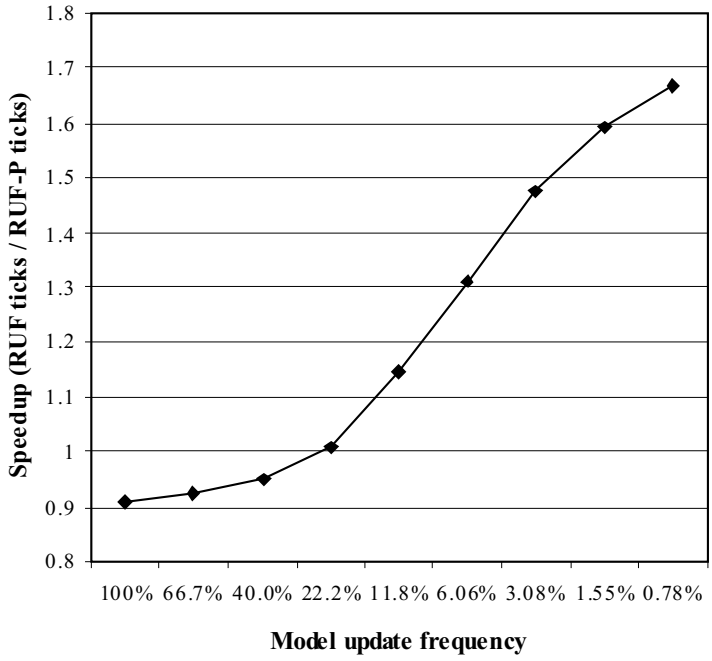

Fig. 5. Fine-grained coding and modeling speedups for various update frequencies

Similar experiments were performed for a couple of other systems described below.

- Sun UltraSparc IIIi (1.06 GHz, 1 MB L2 cache, Solaris 9, Sun C 5.7 compiler). This CPU has 3 integer arithmetic execution units, 2 SIMD execution units, and 160 general-purpose 64-bit integer registers (of which 32 are accessible to a given function) [15].
- AMD Athlon64 3200 in 64-bit mode (2 GHz, 512 kB L2 cache, Linux kernel 2.6, Intel C++ 9.0 and GCC 4.0.0 compilers). This CPU has 4 integer arithmetic execution units, 3 SIMD execution units, 16 general-purpose 64-bit integer registers, 8 multimedia 64-bit registers (for SIMD MMX extensions), and 16 multimedia 128-bit registers (for SIMD SSE2 extensions) [1].
- Intel Xeon DP 3.06 (3.06GHz, 512kB L2 cache, Linux kernel 2.6, Intel C++ 9.0 and GCC 3.3.3 compilers). This CPU has 3 integer arithmetic execution units, 3 SIMD execution units, 8 general-purpose 32-bit integer registers, 8 multimedia 64-bit registers (for SIMD MMX extensions), and 8 multimedia 128-bit registers (for SIMD SSE2 extensions) [8].
- Intel Itanium 2 (1.4 GHz, 256 kB L2 cache, 3 MB L3 cache, Linux kernel 2.4) – as already described, except that we used also GCC 3.2.2 compiler. This CPU has 9 integer arithmetic execution units of which 6 are general-purpose ones, 10 multimedia execution units (capable of SIMD integer processing, 6 of them general-purpose), and 128 general-purpose 64-bit integer registers.

It is worth mentioning that UltraSparc, Athlon64, and Xeon contain smaller number of resources, especially of non-SIMD integer arithmetic execution units potentially useful for fine-grained parallelization, then Itanium 2 (3-4 vs. 9). The Intel and Sun compilers are able to optimize efficiently code placed in different source files; GCC hasn't got such functionality, but it can be simulated by preparing a module that includes all the required files; results reported for GCC are obtained this way. In case of the Itanium we did not analyze whether the speedup is due to MIMD processing or due to SIMD processing. Although all the examined architectures support the SIMD processing, for Itanium and UltraSparc we'd have to reverse engineer binaries to check whether SIMD instructions are actually used. For Athlon64 and Xeon processors, in case of the code compiled using Intel compiler we have a choice whether to use SIMD extensions, gaining access to additional execution units and registers (rows marked "SIMD" in table 1), or to use basic architecture.

Our idea of fine-grained parallelization improved the coding and modeling speed only in the case of Itanium 2 CPU and only for one of the two compilers tested. Clearly, the older version of GCC, as opposed to recent Intel compiler, did not exploit the possibility of fine-grained parallelization. For all the other systems, if the speeds of RUF

and RUF-P variants differ by more than a couple of percents, then the RUF-P variant is slower. As mentioned in section 2 the fine-grained parallelization requires sufficient CPU resources. Probably the simpler CPUs' resources are already fully exploited by the RUF variant. Results obtained on Athlon64 and Xeon by non-SIMD and SIMD are as expected; access to SIMD execution units does not improve the speed of the RUF-P variant.

System CPU	Compiler	Coding and modeling time		
		RUF	RUF-P	speedup
Intel Itanium 2	Intel 9.0	32.9	22.3	1.47
Intel Itanium 2	GCC 3.2.2	39.6	45.2	0.88
Sun UltraSparc IIIi	Sun C 5.7	32.9	38.1	0.86
AMD Athlon64	Intel 9.0	28.0	33.3	0.84
AMD Athlon64	Intel 9.0 SIMD	26.1	33.6	0.78
AMD Athlon64	GCC 4.0.0	27.8	31.5	0.88
Intel Xeon	Intel 9.0	38.4	39.7	0.97
Intel Xeon	Intel 9.0 SIMD	38.2	42.2	0.91
Intel Xeon	GCC 3.3.3	49.5	48.7	1.02

Tab. 1. Coding and modeling time for various systems [ticks per symbol], default update frequency (3.06%)

Stage	RUF	RUF-P
prediction	21%	27%
coding parameter determination and codeword generation	53%	20%
outputting codewords		20%
model updating	12%	16%
reminder (file i/o, etc.)	14%	17%

Tab. 2. Execution time of stages of the compression process (time relative to the time of a whole compression process)

As mentioned in section 2, the RUF-P method also permits certain medium-grained parallelizations. We measured the execution time of various stages of the compression process. In Table 2 we report the time expressed in percents of the time required to perform the complete compression process including the file i/o and the prediction. Results were obtained on the Itanium 2 processor using executables compiled by Intel compiler and for the default 3.08% model update frequency. Except for the model update time, results were calculated based on measurements of compression process with skipping of certain stages. Model update time was estimated based on comparing results for differ-

ent model update frequencies; the update time may include overhead of initializing the processing of a block of symbols and thus may be overstated at the expense of time of coding parameters determination and codewords generation.

For the RUF variant we assume that the model update may be performed in parallel to coding parameter determination and codeword generation and to outputting of codewords since actual implementation is as good for it (Fig. 4) as the RUF-P. From results presented in Table 2, we can expect for the RUF variant the medium-grained parallelization speedup of no more than about 2.

For the RUF-P variant the figures in Table 2 are relative to the speed already increased by the fine-grained parallelization. Let's estimate what further speedup can be achieved by employing medium-grained parallelism. The prediction stage itself may be parallelized, e.g., it may be performed in parallel for halves of a row of image pixels, so it does not limit attainable speedups. The speedup is now limited by outputting of codewords – the slowest inherently serial part of the compression process. It is also limited by the coding parameter determination and codeword generation, i.e., the stage that in the RUF-P variant may run in parallel to outputting of codewords. Note, that this stage can be performed independently for two halves of a block of symbols, but in practice the thread synchronization costs may thwart parallelization effects since the block size is small and variable (for the update frequency used, it is in the range [1, 64], 32.5 on average). Now, the slowest stages that limit the speedup attainable through medium-grained parallelization require about 20% of the time of a whole compression process. Hence for the RUF-P variant we can expect the medium-grained parallelization speedup of no more than 5.

4. Conclusions

Modeling and coding is the most complex part of many adaptive compression algorithms; it is also an inherently serial process. The method of reduced model update frequency is a modification of the typical adaptive scheme, which was originally employed in the SFALIC lossless image compression algorithm in order to improve the speed of modeling at the cost of a negligible worsening of the modeling quality.

In this paper, we notice that the above method permits to parallelize the compression process. We experimentally evaluate the fine-grained parallelization speedups and estimate the medium-grained parallelization effects. For the Itanium 2 CPU and a recent Intel compiler, the fine-grained parallelization improved the coding and modeling speed of SFALIC by about 50%. For other architectures and for various compilers no significant speedups were observed indicating that the speedup is conditioned on both the CPU architecture and the compiler. Reduced update frequency method also allows greater extent of medium-grained parallelism since it allows splitting, into two threads, the slowest

part of the compression process (and accelerates it through fine-grained parallelism as well). The presented method may be used for parallelizing some other adaptive algorithms in which it is applicable, provided that we may accept worsening of the modeling quality.

Acknowledgments

This research was supported by the grant BK-239/Rau-2/2005 from the Institute of Computer Science, Silesian University of Technology.

References

[1] Advanced Micro Devices: AMD64 Architecture Programmer's Manual Volume 1: Application Programming. Rev. 3.11, December 2005.

[2] Allan V.H., Jones R.B., Lee R.M., Allan S.J.: Software Pipelining. ACM Computing Surveys, 27(3), 1995, pp. 367-432.

[3] Carpentieri B., Weinberger M.J., Seroussi G.: Lossless compression of Continuous-Tone Images. Proceedings of the IEEE, 88(11), 2000, pp. 1797-809.

[4] Flynn M.J.: Very high-speed computing systems. Proceedings of the IEEE, 54(12), 1966, pp. 1901-9.

[5] Golomb S.W.: Run-Length Encodings. IEEE Trans. on Information Theory, IT-12, 1966, pp. 399-401.

[6] Howard P.G., Vitter J.S.: Fast and efficient lossless image compression. Proceedings DCC '93 Data Compression Conference, IEEE Comput. Soc. Press, Los Alamitos, CA, 1993, pp. 351-60.

[7] Huffman S.A.: A method for the construction of minimum-redundancy codes. Proceedings of the Institute of Radio Engineers, 40(9), 1952, pp. 1098-101.

[8] Intel: IA-32 Intel Architecture Optimization Reference Manual. December 2003.

[9] Intel: Intel Itanium 2 Processor Reference Manual For Software Development and Optimization. April 2003.

[10] Moffat A., Neal R.M., Witten I.H.: Arithmetic Coding Revisited. ACM Transactions on Information Systems, 16(3), 1998, pp. 256-94.

[11] Rice R.F.: Some practical universal noiseless coding techniques – part III. Jet Propulsion Laboratory tech. report JPL-79-22, 1979.

[12] Shannon C.E.: A Mathematical Theory of Communication. Bell System Technical Journal, 27, 1948, pp. 379-423, 623-56.

[13] Starosolski R.: Performance evaluation of lossless medical and natural continuous tone image compression algorithms. Proc. SPIE, 5959, 2005, pp. 116-27.

[14] Starosolski R.: Simple Fast and Adaptive Lossless Image Compression Algorithm. Software-Practice and Experience, 37(1), 2007, pp. 65-91.

[15] SUN Microsystems: UltraSPARC III Cu User's Manual. Version 2.2.1, January 2004.

Zrównoleglenie adaptacyjnego algorytmu kompresji z użyciem metody zmniejszonej częstości aktualizacji modelu danych

Streszczenie

Modelowanie i kodowanie to najbardziej złożone elementy wielu adaptacyjnych algorytmów kompresji, przy czym sam proces kompresji oparty o modelowanie i kodowanie musi być realizowany w sposób sekwencyjny. Metoda zmniejszonej częstości aktualizacji modelu danych to modyfikacja zastosowana do adaptacyjnego algorytmu kompresji, aby poprawić prędkość modelowania kosztem nieznacznego, z praktycznego punktu widzenia, pogorszenia jakości modelowania. W niniejszej pracy zauważono, iż zastosowanie tej metody umożliwia zrównoleglenie algorytmu kompresji. Badania eksperymentalne wykazały, że dzięki wykorzystaniu równoległości drobnoziarnistej dla procesora Itanium 2 i algorytmu SFALIC metoda zmniejszonej częstości aktualizacji modelu danych pozwala na zwiększenie prędkości kodowania i modelowania o około 50%. Przeprowadzone szacunki pokazały, że metoda ta pozwala również na wykorzystanie równoległości średnioziarnistej znacznie większym stopniu.