Textual Data Compression Speedup by Parallelization

GORAN MARTINOVIC, CASLAV LIVADA, DRAGO ZAGAR
Faculty of Electrical Engineering
Josip Juraj Strossmayer University of Osijek
Kneza Trpimira 2b, 31000 Osijek
CROATIA
goran.martinovic@etfos.hr, caslav.livada@etfos.hr, drago.zagar@etfos.hr, www.etfos.hr

Abstract: - When the omnipresent challenge of space saving reaches its full potential so that a file cannot be compressed any more, a new question arises: “How can we improve our compression even more?”. The answer is obvious:”Let´s speed it up!”. This article tries to find the meeting point of space saving and compression time reduction. That reduction is based on a theory in which a task can be broken into smaller subtasks which are simultaneously compressed and then joined together. Five different compression algorithms are used two of which are entropy coders and three are dictionary coders. Individual analysis for every compression algorithm is given and in the end compression algorithms are compared by performance and speed depending on the number of cores used. To summarize the work, a speedup diagram is given to behold if Mr. Amdahl and Mr. Gustafson were right.

Key-Words: - data compression, lossless coding, entropy coder, dictionary coder, parallel computing, compression time speed up

1 Introduction

According to [1], compression is used to reduce physical space needed for storing data using certain methods of data writing. The main unit for data storage is a file and depending on the data type within the file, some repetition is noticed. The data is stored once and after that only the position of repeated data is stored. In this way, it is possible to decrease space needed for data storage, which depends on data type and structure.

There are two kinds of compression: lossless and lossy compression (compression without and with certain data loss). Lossless compression is carried out by data compression algorithms which ensure that the reconstructed data from compressed files corresponds to the starting data. Lossless compression is used when it is necessary to save the original data and typical examples by which lossless compression must be used are executable files, source codes, textual data, etc. [2], [4].

The majority of programs performing lossless compression practice two steps on the data that will be compressed and these steps are:

- Generation of input data statistical model
- Input data can be transformed into sequences of bits by using the statistical model so that less frequently occurring data give longer output sequences [5].

There are two main statistical models: static and adaptive model. In the static model, data is analyzed, the model is created and packed together with compressed data. This approach is very simple, but there is a chance that the model itself takes a lot of space in the compressed file. Another problem is that one model is used for all input data and it can give bad results on files containing various data [1].

Adaptive models dynamically refresh the model as the data is compressed. Coder and decoder start with an elementary model that gives bad compression for the input data but as the model learns more about the input data, compression efficiency increases. The majority of compression methods use adaptive models.

One lossless compression algorithm cannot efficiently compress all possible data, and completely random data streams cannot be compressed. For this reason, there are various algorithms designed for certain file types.

By using lossy compression, the reconstructed data differs from the input data, but these differences are so small and the reconstructed data is still usable. According to [1] and [2], lossy compression methods are used for multimedia compression (audio, video and pictures).

The original file contains a certain amount of information and there is a lower data size boundary able to carry the overall information. In most cases files or data streams contain more information than needed, e.g. a picture can have more details than a human eye can distinguish, and lossy compression methods tend to exploit imperfections of human perception [2].

Two main lossy compression schemes are:

- Lossy transformation codecs (coder-decoder) – Picture or audio samples are taken, divided into smaller parts, quantized and compressed using entropy coding [1].

- Predictive lossy codecs – preceding and/or following decoded data is used to assume a current picture or audio sample. The difference between the assumed and the real sample with additional data is quantized and coded.

2 Compression Methods Analyzed

Since we deal with textual data, only the lossless compression methods will be shown and analyzed. These methods are: Huffman coding, Arithmetic coding, LZ77, LZ78 and LZW [3].

2.1 Huffman Coding

As in [2], Huffman coding codes symbols with variable length code words, depending on their probability of incidence.

Huffman coding is based on two simple facts:

1. In optimal code, symbols with greater probabilities of incidence have smaller code words than symbols with lesser probabilities of incidence.
2. In optimal code, two symbols with the smallest probabilities have code words of the same length.

Huffman coding algorithm is shown in Fig. 1, similarly to [2]:

1. *Sort symbols by falling probabilities.*
2. *Link two symbols with least probabilities into one new symbol, on branching assign "0" to one branch and "1" to the second branch.*
3. *Sort the obtained symbols by probabilities.*
4. *Repeat until there is only one symbol.*
5. *Go back through tree read codes.*

Fig. 1 - Huffman coding algorithm

Algorithm data structure is a binary tree. The decoding algorithm uses the same procedure as the coder for building a tree and before sending data, the coder must first send the probabilities of symbol incidence.

Advantages of the Huffman algorithm:

1. Easy implementation.
2. Near optimal coding is achieved for "good" probabilities of incidence.

Disadvantages of the Huffman algorithm:

1. Probabilities of incidence must be known.
2. Codes can become "bad" for wrong probabilities of incidence.

A coding example will be given in the next section: We can assume that we have the following stream – AACBBABCAA.

First, we must calculate the probabilities of each symbol as shown in Table 1. The coding tree can then be created by using the data from Table 1, which is shown in Fig 2.

Table 1. Symbol probabilities

Symbol	Probability
A	0.5
B	0.2
C	0.3

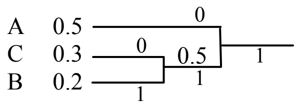

Fig. 2 - Huffman coding example

The binary code of every symbol can be acquired by reading the Huffman tree in right to left manner and the resulting code can be seen in Table 2.

Table 2. Output code

Symbol	Binary Code
A	10
B	111
C	110

Using Table 2 a unique code can be written for the input stream as follows: 1010110111111101111101010.

2.2 Arithmetic Coding

The algorithm takes queues of symbols as the input data (message) and transforms them into a floating point number, depending on a known statistical model. Arithmetic coding is shown in Fig. 3, as in [2]:

1. *Divide interval [0,1] into M intervals that correspond to symbols, symbol length is proportional to symbols probability.*
2. *Select the interval of the first symbol in queue.*
3. *Subdivide the current symbol interval into new M subintervals, proportional to their probabilities.*
4. *From these subintervals, select the one that matches the next symbol in queue.*
5. *Repeat steps 3 and 4 until all symbols are coded.*
6. *Output the interval value in binary form.*

Fig. 3 - Arithmetic coding algorithm

The procedure of interval division in arithmetic coding is shown in Fig. 4.

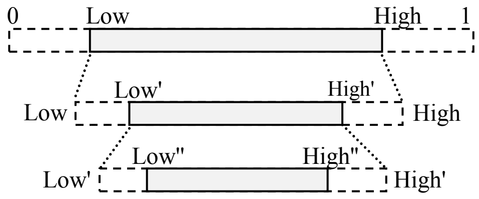

Fig. 4 - Arithmetic coding algorithm

The arithmetic coding procedure can be shown on the following example: Assume that we have the following input stream – BBAB. First, we must define the ranges and the probability line. The ranges and probabilities can be seen in Table 3.

Table 3. Symbol probabilities and ranges

Symbol	Probability	Range
A	0.25	[0, 0.25)
B	0.75	[0.25, 1)

Our input stream is BBAB and we can calculate its output number. The coding process can be seen in Table 4.

Table 4. Output number calculation

Symbol	Range	Low value	High value
		0	1
B	1	0.25	1
B	0.75	0.4375	1
A	0.5625	0.4375	0.578125
B	0.140625	0.47265625	0.578125

According to Table 4, the output number will be 0.47265625.

2.3 LZ77

Algorithm LZ77 uses a code word dictionary and tries to replace a stream of symbols with the reference to the stream dictionary location. The main assumption is that the reference to the stream in the dictionary is shorter than the stream itself. The stream is coded using an arranged pair length-distance which can be described as follows: "every *length* symbol equals the symbol that is exactly *distance* symbols after uncompressed stream". Coder and decoder must monitor the number of recent data, last 2 kB, 4 kB or 32 kB. The structure in which this data is held is called a *sliding window;* hence, LZ77 is sometimes called sliding window compression. Coder needs this data to discover if there is any correspondence and decoder needs this data for reference interpretation ([6] and [7]).

The coding algorithm is shown in Fig. 5, according to [1]:

1. *Initialize the dictionary to a known value.*
2. *Read an uncoded string that is the length of the maximum allowable match.*
3. *Search for the longest matching string in the dictionary.*
4. *If a match is found to be greater than or equal to the minimum allowable match length:*
    a. *Write the encoded flag, then the offset and length to the encoded output.*
    b. *Otherwise, write the uncoded flag and the first uncoded symbol to the encoded output.*
5. *Shift a copy of the symbols written to the encoded output from the unencoded string to the dictionary.*
6. *Read a number of symbols from the uncoded input equal to the number of symbols written in Step 4.*
7. *Repeat from Step 3, until all the entire input has been encoded.*

Fig. 5 - LZ77 coding algorithm

We can review the LZ77 coding algorithm on the next data stream: CBCABBBCA. The coding process as well as the output can be seen in Table 6 and the graphical representation of the LZ77 coding process of the above mentioned input stream is shown in Fig 6.

Table 5. Input data stream

Position	1	2	3	4	5	6	7	8	9
Character	C	B	C	A	B	B	B	C	A

Table 6. Coding process

Step	Position	Match	Character	Output
1.	1	--	C	(0,0) C
2.	2	--	B	(0,0) B
3.	4	C	A	(2,1) A
4.	5	B	B	(3,1) B
5.	7	B C	A	(5,2) A

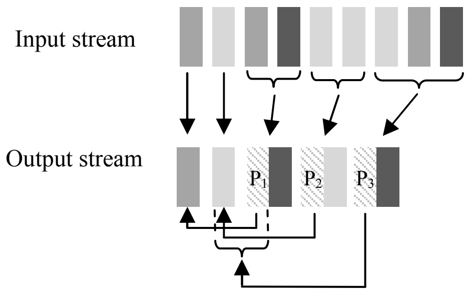

Fig. 6 - LZ77 algorithm

2.4 LZ78

LZ77 algorithm works with previous data and LZ78 tries to work with future data. LZ78 reads input stream in advance and compares it with the dictionary which is constantly updated. Input stream will be read until the corresponding data in the dictionary is found and in that moment, the position and length of data are written into dictionary [7].

The LZ78 coding algorithm is shown in Fig. 7, according to [1]:

```
1. Empty the dictionary and prefix p.
2. For every c (character from an uncompressed file)
    a. If (p + c) exists in the dictionary, then p = p + c.
    b. Else, output the dictionary code for p to output and output c, add (p + c) to the dictionary and empty p.
3. Add the dictionary code for p to output.
```

Fig. 7 - LZ78 coding algorithm

The following input stream will be used to show the LZ78 coding process: ABBCBCABA. The coding algorithm shown in Fig. 7. is used on the before mentioned input stream and the resulting coding process and the output are displayed in Table 8.

Table 7. Input data stream

Position	1	2	3	4	5	6	7	8	9
Character	A	B	B	C	B	C	A	B	A

Table 8. Coding process

Step	Position	Dictionary	Output
1.	1	A	(0, A)
2.	2	B	(0, B)
3.	3	B C	(2, C)
4.	5	B C A	(3, A)
5.	8	B A	(2, A)

The biggest advantage over the LZ77 is a reduced number of string comparisons in each encoding step. LZ77 and LZ78 have a similar compression ratio. The LZ77 coding algorithm is shown in Fig 8.

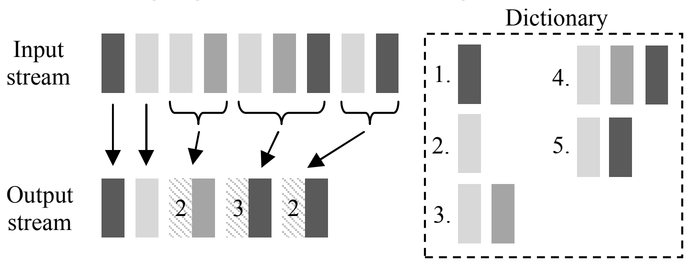

Fig. 8 - LZ78 algorithm

2.5 LZW Coding

LZW is a refined version of the LZ78 algorithm; hence, it is faster but usually not optimal because of the constrained data analysis. The LZW coding algorithm is shown in Fig. 9, as in [8]:

```
1. Set w to zero.
2. For every c (character from an uncompressed file)
    c. If (w + c) exists in the dictionary, then w = w + c.
    d. Else, add a dictionary code for w to output, add (w + c) to the dictionary and set w = c.
3. Add the dictionary code for w to output.
```

Fig. 9 - LZW coding algorithm

We can see the LZW coding efficiency on a next example: The alphabet consists of three symbols – A, B, C which constitute the starting dictionary shown in Table 9.

Table 9. LZW starting dictionary

Location	Character
(1)	A
(2)	B
(3)	C

Input stream that needs to be coded and the LZW algorithm execution are shown in Table 10 and Table 11, respectively.

Table 10. Input data stream

Position	1	2	3	4	5	6	7	8	9
Character	A	B	C	B	C	A	B	C	A

Table 11. LZW algorithm execution

Step	Position	Dictionary content		Output
--	--	(1)	A	--
--	--	(2)	B	--
--	--	(3)	C	--
1.	1	(4)	A B	(1)
2.	2	(5)	B C	(2)
3.	3	(6)	C B	(3)
4.	4	(7)	B C A	(5)
5.	6	(8)	A B C	(4)
6.	8	(9)	C A	(3)
7.	--	--	--	(1)

The main advantage of LZW over LZ77-based algorithms is in the speed because there aren't that many string comparisons to perform [9].

3 Parallel Processing

Computer software has been written for serial computation and to solve a problem, an algorithm is constructed which produces a serial stream of instructions. These instructions are executed on the central processing unit on one computer. Only one instruction may be executed at a given time and when that instruction is finished, the next one is executed.

Parallel processing implies simultaneous execution of multiple instructions. It is based on the principle that bigger tasks can always be divided into smaller ones which are simultaneously executed. Communication and synchronization between various subtasks are the biggest restrictions for achieving good parallel processing performance [8].

Theoretically, by doubling processing elements, execution time should halve, and with the second doubling, execution time should halve again. However, a few parallel algorithms achieve this optimal speed-up. The majority of parallel algorithms have this optimal (linear) speed-up for a small number of processing elements but with a further increase of processing elements execution time becomes constant [10].

3.1 Amdahl`s law and Gustafson`s law

The performance of an algorithm on a parallel computing platform depends on parallelizing the algorithm to achieve performance so it is important to be aware of Amdahl's law, originally formulated by Gene Amdahl in the 1960's. It states that a small portion of the program which cannot be parallelized will limit the overall speedup available from parallelization. Any large math or engineering problem will typically consist of several parallelizable parts and several non-parallelizable (sequential) parts. This relationship is given by Amdahl's law:

$$S = \frac{1}{1-P}, \tag{1}$$

where S is the speedup of the program (as a factor of its original sequential runtime), and P is the fraction that is parallelizable. If the sequential portion of a program is 20% of the runtime, we can get no more than a 5x speedup, regardless of how many processors are added. This puts an upper bound on the usefulness of adding more parallel execution units. A graphical representation of Amdahl`s law is given in Fig. 10.

A task has two independent parts, A and B. B takes 20% of the time of the whole process. We may be able to make this part 4 times faster, but this insignificantly reduces the time for the whole computation. In contrast to that, one may need to perform less work to make part A be twice as fast. This will make the computation much faster than by optimizing part B, even though B gets a greater speed-up, (4x versus 2x).

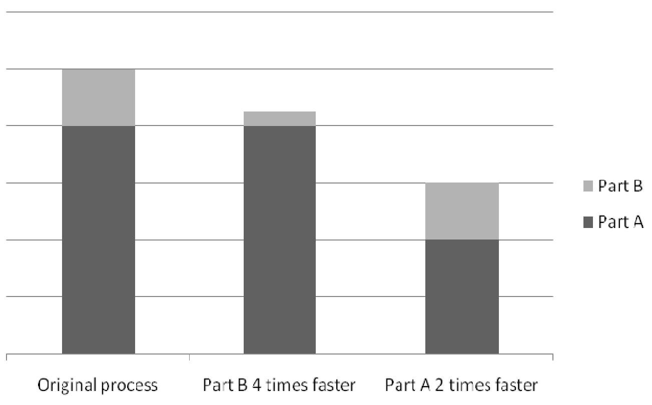

Fig. 10 – Amdahl`s law

Gustafson's law is another law in computer engineering, closely related to Amdahl's law. Gustafson's law can be formulated as:

$$S(P) = P - \alpha(P-1), \tag{2}$$

where P is the number of processors, S is the speedup, and α the non-parallelizable part of the process. Amdahl's law assumes a fixed-problem size and that the size of the sequential section is independent of the number of processors, whereas Gustafson's law does not make these assumptions [10].

Not every parallelization will result in execution time speed-up. As the tasks are divided in more and more threads, these threads need more time for mutual communication. It is possible for this additional communication to dominate the time needed to solve problems; therefore, further parallelization increases the program execution time, instead of decreasing it. This phenomenon is called parallel slowdown [11].

3.2 Parallel programming approaches

As already mentioned, the majority of programs are sequential and have a single line of control, and in order to make many processors work on a single program, a program must be divided into smaller independent chunks so that each processor can work on separate chunks. The most prominent parallel programming approaches are:

- Data parallelism,
- Process parallelism, and
- Farmer and worker model.

In case of data parallelism, the divide-and-conquer technique is used to split data into multiple sets and each data set is processed by a different processor by using the same instruction. This approach is very suitable for processing on a machine on the SIMD model. In case of process parallelism, a given operation has multiple activities which can be processed on multiple processors. In case of the farmer and worker model, job distribution is used; one processor is configured as master and all remaining processors are designated as slaves. The master assigns jobs to slaves and they inform the master on completion, which in turn collects results [11].

In this research we used a multi-core processor which consists of several processing units (cores) that can execute more instructions per cycle from different instruction streams, i.e. we used the data parallelism approach.

4 Parallel Processing Data Compression

Data compression program will be executed on a multi-core processor computer, i.e. on a computer with four cores. Computer configuration is as follows: CPU: Intel Core i5 750 @ 2.67 GHz, RAM: 4 GB, Windows 7 OS. The coding algorithm on a multi-core system can be divided into several steps [10]:

- read input data stream,
- separate input data stream in N equal parts where $N \leq$ the number of processor cores,
- compress disjointed data using one of the mentioned compression methods,
- join compressed parts into one file.

Compression program was written in C++ for command prompt similar to [12] and [15]. An executable file needs additional parameters, like compression type, input file and output file. Compression was repeated ten times, so that we could obtain the best results because various processes were running in the background of the computer. After that, statistical data about compression was written in a file, from where we did our analysis. Fig. 11 shows data compression using four cores similarly to [13].

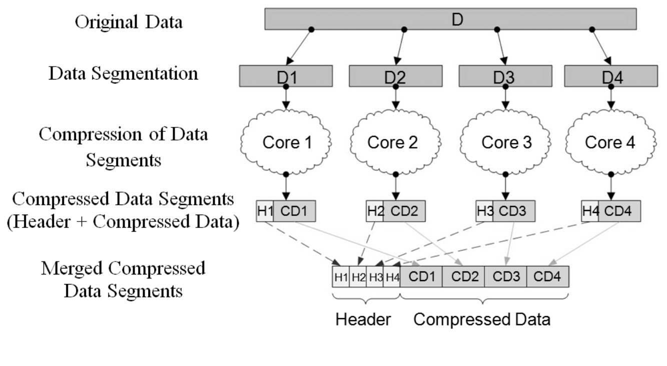
Original Data
D
Data Segmentation
D1
D2
D3
D4
Compression of Data Segments
Core 1
Core 2
Core 3
Core 4
Compressed Data Segments (Header + Compressed Data)
H1 CD1
H2 CD2
H3 CD3
H4 CD4
Merged Compressed Data Segments
H1 H2 H3 H4 CD1 CD2 CD3 CD4
Header
Compressed Data

Fig. 11 - Multi-core data compression algorithm

This parallel algorithm can only go as fast as its slowest parallel work, as shown in Fig. 12.

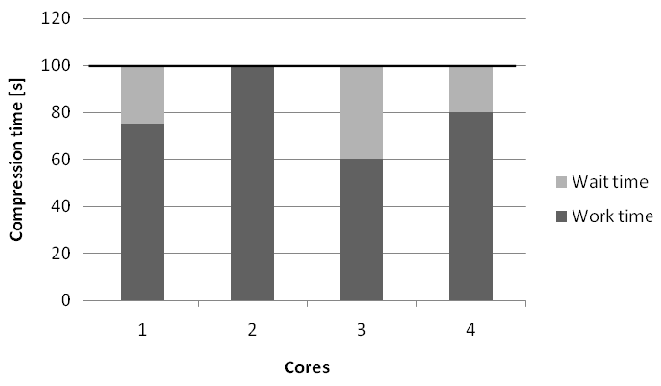

Fig. 12 – Parallel work on different cores

In order to have a successful and correct compression, all cores must finish their individual parallel work. If one or more cores finish their parallel work before other cores, they must wait. In an example shown in Fig. 12, cores 1, 3 and 4 must wait until core 2 finishes its work.

5 Results and Analysis

The data compression algorithm analysis will be tested on a textual file shown in Table 12. The number of compression iterations is ten. Results are analyzed and minimum, maximum and mean values are shown in tables.

Table 12. Source file

Document name	Size [in bytes]	Description
text.txt	5,910,599	Text

5.1 Time and Average Time of File Compression Using Arithmetic Coding

Table 13 shows mean and minimum values of the time needed for compression, compressed file size and the achieved compression. The values were acquired by analyzing results shown in Fig. 13.

Table 13. Compression times by using arithmetic coding

Time [ms]	Number of cores				Compressed file size [byte]	Compression ratio [%]
	1	2	3	4		
min	2730	1358	920	780	3,530,681	59.73
avg	2753	1365	922	832		

The compression ratio for the implemented compression method was calculated by using expression (3).

$$Compression\ ratio[\%] = \frac{Compressed\ file\ size}{Original\ file\ size} * 100\% \qquad (3)$$

From Table 13, but also from Fig. 13 we can conclude that compression time decreases with an increase in a utilized core number. By doubling the number of cores, compression time halves. Compressed data is stored in the file *bible_arit.txt.*

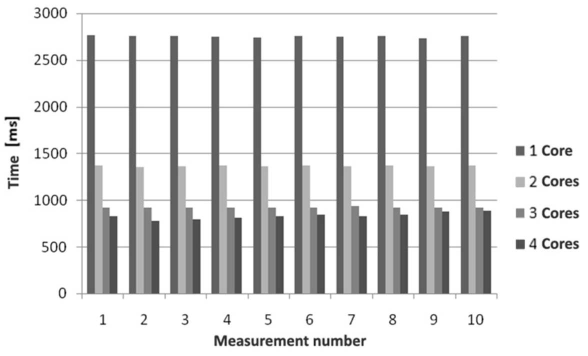

Fig. 13 - Compression time comparison depending on the number of cores used in arithmetic coding

5.2 File Compression Times Using Huffman Coding

Table 14 shows compression times and Fig. 14 enables a comparison of compression times when using Huffman coding.

Table 14. Compression times by using Huffman coding

Time [ms]	Number of cores				Compressed file size [byte]	Compression ratio [%]
	1	2	3	4		
min	2995	1482	998	827	3,671,779	62.12
avg	3016	1495	1009	839		

As well as in arithmetic coding, in Huffman coding compression time decreases proportionally with an increase in the number of cores. The compressed file size created by Huffman coding is larger than the compressed file size generated by arithmetic coding. Compression time is longer in Huffman coding than in arithmetic coding.

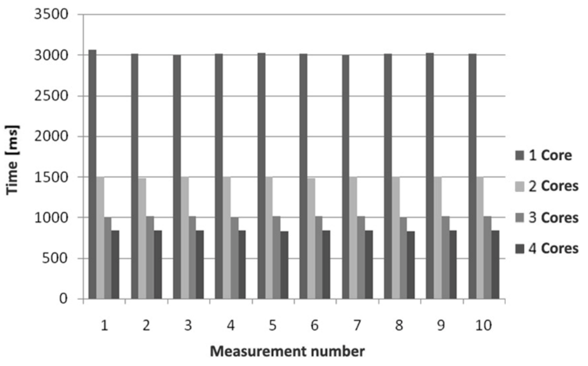

Fig. 14 - Compression time comparison depending on the number of cores used in Huffman coding

5.3 Core Number Changing Effect on Data Compression Using LZ77

Table 15 and Fig. 15 show compression time when using LZ77 coding and the comparison of compression times depending on the number of cores, respectively.

Table 15. Compression times by using LZ77 coding

Time [ms]	Number of cores				Compressed file size [byte]	Compression ratio [%]
	1	2	3	4		
min	199649	101684	69326	57096	2,642,705	44.71
avg	199631	101434	69250	56901		

Data compression time reduces when the number of cores increases. Compression time difference is very small for cases when three and four cores are used.

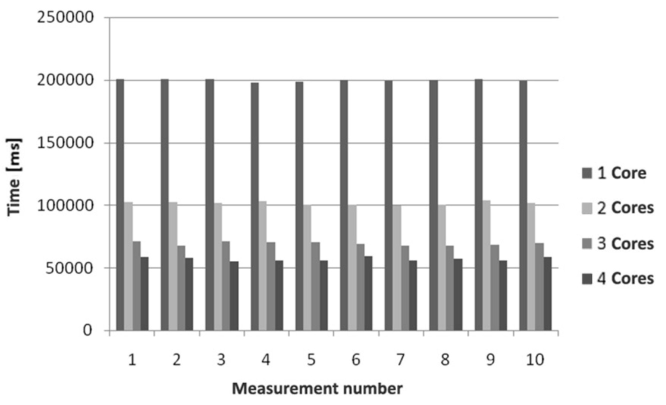

Fig. 15 - Compression time comparison depending on the number of cores used in LZ77 coding

5.4 Results of LZ78 Coding Depending on the Number of Cores Used

Table 16 and Fig. 16 show compression time when using LZ78 coding and the comparison of compression times depending on the number of cores, respectively.

Table 16. Compression Times by Using LZ78 Coding

Time [ms]	Number of cores				Compressed file size [byte]	Compression ratio [%]
	1	2	3	4		
min	3916	3338	3042	3042	3,140,033	53.13
avg	5148	3356	3062	3059		

Data compression time reduces when the number of cores increases. Compression time difference is very small for cases when three and four cores are used.

5.5 Results of LZW Coding Depending on the Number of Cores Used

Table 17 shows average and minimum values of file compression times, compressed file size and the compression ratio. The values were acquired by analyzing results shown in Fig. 17.

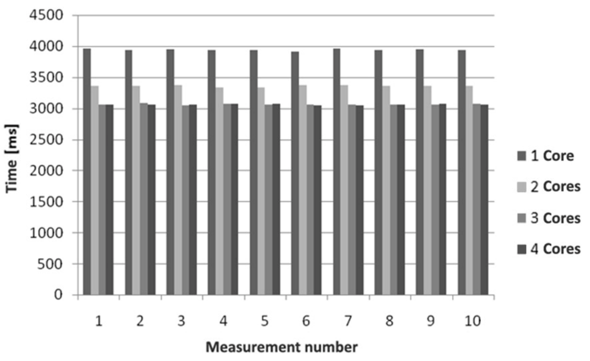

Fig. 16 - Compression time comparison depending on the number of cores used in LZ78 coding

Table 17. Compression times by using LZW coding

Time [ms]	Number of cores				Compressed file size [byte]	Compression ratio [%]
	1	2	3	4		
min	5679	4431	4388	4368	2,434,644	41.19
avg	5724	4502	4462	4379		

LZW coding achieves the best compression ratio but compression time only increases with the first core doubling and with a further core number increase compression time does not change much.

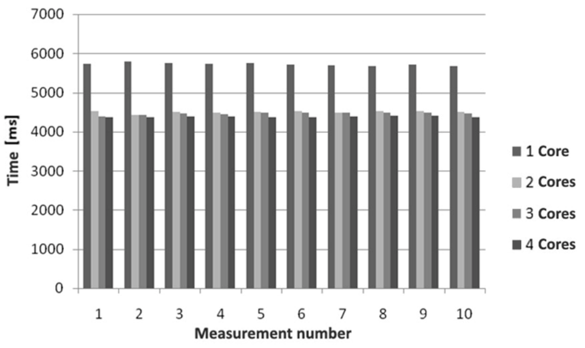

Fig. 17 - Compression time comparison depending on the number of cores used in LZW coding

5.6 Comparative analysis of minimal data compression times

Table 18 shows compression time (minimum values in ms) and the compression ratio of the file text.txt for all coding methods.

Table 18. Compression times for various coding methods depending on the number of cores used

Compression method	Number of cores				Compression ratio [%]
	1	2	3	4	
Arithmetic	2730	1358	920	780	59.73
Huffman	2995	1482	998	827	62.12
LZ78	3916	3338	3042	3042	53.13
LZW	5679	4431	4388	4368	41.19

Fig. 19 shows file sizes (in bytes) after compressing with a certain coding method.

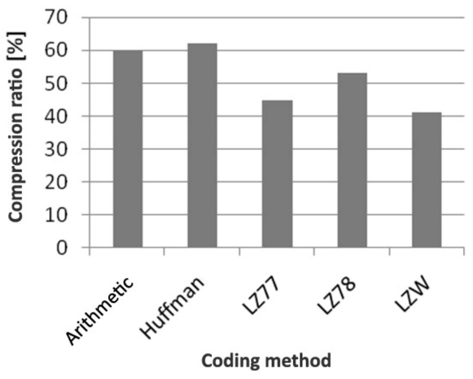

Fig. 18 - Compression ratio of all coding methods used

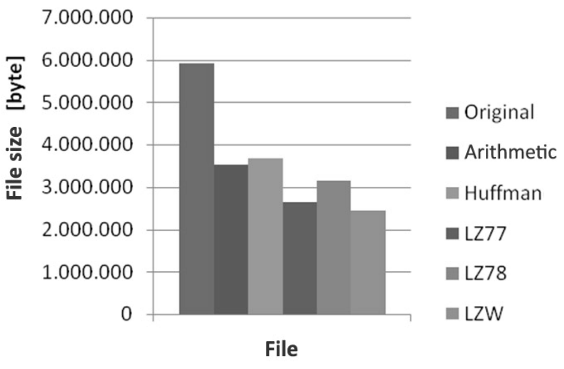

Fig. 19 - Compressed file size comparison depending on the coding method used

Fig. 20 shows compression times of all algorithms and because of LZ77, a logarithmic scale is used for presentation of compression times.

Fig. 21. shows the achieved speedup by increasing the number of cores. Speedup is calculated by using expression (4) where $n = 2, 3, 4$, as in [14].

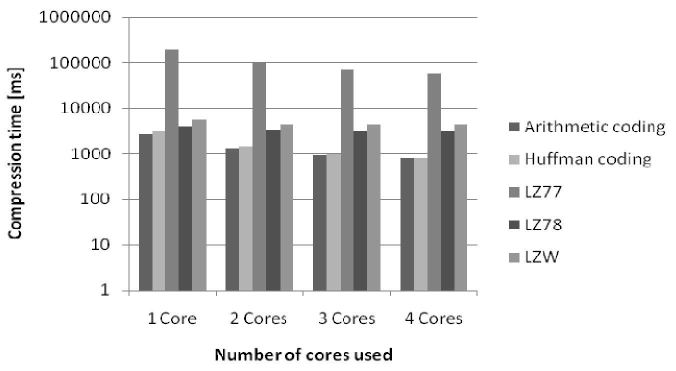

Fig. 20 - Compression times for all used methods

$$Speedup = \frac{Compression\ time\ on\ n\ cores}{Compression\ time\ on\ one\ core} \quad (4)$$

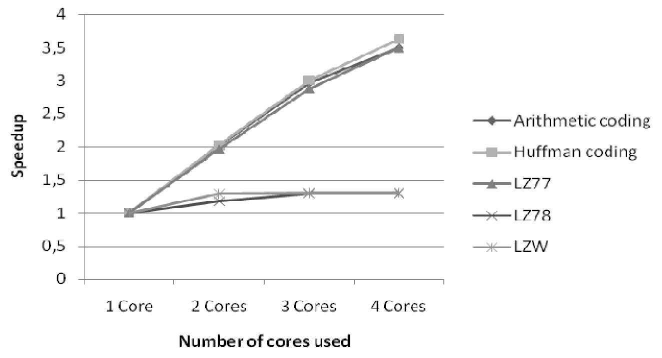

Fig. 21 – Achieved speedup

6 Conclusion

The goal of this paper was to see how much the parallelization process shortens the compression time of files consisting only of textual data. The parallelization process includes splitting up the task in several smaller subtasks and their simultaneous execution. This was achieved by using multiple cores. A logical thing to do was to conclude that the core number increment would lead to a compression time decrease. Our former conclusion was proven from the results of this paper. We used two types of coders in this paper: entropy coders and dictionary coders. Dictionary coders are inferior to entropy coders as to compression time but superior as to file size reduction. The compression time speedup for entropy coders was close to linear speedup but the discrepancies were due to non-parallelizable parts of compression. Dictionary coders, LZ78 and LZW have more non-parallelizable parts due to the dictionary so their speedup is significantly smaller than the entropy coders` speedup, being close to 1. According to what we have seen so far, we can securely conclude that the parallelization procedure had the best impact on entropy coders, especially on arithmetic coding.

Acknowledgment

This work was supported by research project grant No. 165-0362980-2002 from the Ministry of Science, Education and Sports of the Republic of Croatia.

References:

[1] X1. M. Pu, *Fundamental Data Compression*, Butterworth-Heinemann, 2005.

[2] X2. K.P. Subbalakshmi, Lossless Image Compression, *Lossless Compression Handbook*, Academic Press, Chapter 9, 2003, pp. 207-226.

[3] X3. T. Moffat, C. Bell, I. H. Witten, Lossless Compression for Text and Images, *Int. J. of High Speed Electronics and Systems*, Vol.8, No.1, 1997, pp. 179-231.

[4] X4. J. Platoš, V. Snášel, Compression of small text files, *Advanced Engineering Informatics*, Vol.22, Issue 3, 2008, pp. 410-417.

[5] X5. J. Ziv, A. Lempel, A Universal Algorithm for Sequential Data Compression, *IEEE Transaction on Information Theory*, Vol.23, No.3, 1977, pp. 337-343.

[6] X6. S. C. Sahinalp, N. M. Rajpoot, Dictionary-Based Data Compression: An Algorithmic Perspective, *Lossless Compression Handbook*, Academic Press, Ch.6, 2003, pp. 153-167.

[7] X7. F. Rizzo, J. A. Storer, B. Carpentieri, LZ-based image compression, *Information Sciences*, Vol.135, Issues 1-2, 2001, pp. 107-122.

[8] X8. M. R. Nelson, LZW Data Compression, *Dr. Dobb`s Journal*, Vol.14, Issue 10, 1989, pp. 29-39.

[9] X9. R.Logeswaran, Enhancement of Lempel-Ziv Coding Using a Predictive Pre-Processor Scheme for Data Compression, *Advances in Information Science and Soft Computing*, WSEAS Press, 2002, pp. 238-243.

[10] X10. A. Grama, G. Karypis, V. Kumar, A. Gupta, *Introduction to Parallel Computing*, Addison Wesley, 2nd Ed., 2003.

[11] X11. J. A. Storer, J. H. Reif, A parallel architecture for high speed data compression, *Journal of Parallel and Distributed Computing*, Vol.13. Issue 2, 1991, pp. 222-227.

[12] X12. W. Bielecki, D. Burak, Parallelization of the AES Algorithm, *Proc. of the 4th WSEAS Int. Conf. on Information Security, Communications and Computers, Tenerife, Spain*, December 16-18, 2005, pp. 224-228.

[13] X13. I. Aziz, N. Haron, L.T. Jung, W. R.W. Dagang, Parallelization of Prime Number Generation using Message Passing Interface, *WSEAS Trans. on Computers*, Vol. 7, Issue 4, 2008, pp. 291-303.

[14] X14. H. P. Flatt, K. Kennedy, Performance of parallel processors, *Parallel Computing*, Vol.12, Issue 1, 1989, pp. 1-20.

[15] X15. R. Bose, An Efficient Method to Calculate the Free Distance of Convolutional Codes, *Proc. of the 6th WSEAS Int. Conf. on Electronics, Hardware, Wireless and Optical Communications, Corfu Island, Greece*, February 16-19, 2007, pp. 60-63.