Burrows-Wheeler Post-Transformation with Effective Clustering and Integer Coding Through Vectorization

Amit Kumar Yadav (Kumaramit.ak2019@gmail.com)
Department of Electronics and Computer Engineering, Pulchowk Campus, IOE, TU, Nepal
Sanjeeb Prasad Panday (sanjeeb@ioe.edu.np)
Department of Electronics and Computer Engineering, Pulchowk Campus, IOE, TU, Nepal

Research Article

Keywords: BWT, Integer Compression, Lossless Compression, Non-parametric coding, Non-statistical Coding, Run Length Encoding, Move to Front coding, Vectorization

Posted Date: May 16th, 2022

DOI: https://doi.org/10.21203/rs.3.rs-1655438/v1

Burrows-Wheeler Post-Transformation with Effective Clustering and Integer Coding Through Vectorization

Amit Kumar Yadav[1*] and Dr Sanjeeb Prasad Panday[1*]

[1*]Department of Electronics and Computer Engineering, Pulchowk Campus, IOE, TU, Kathmandu, Nepal.

*Corresponding author(s). E-mail(s): kumaramit.ak2019@gmail.com; sanjeeb@ioe.edu.np;

Abstract

This research is mainly based on lossless integer Compression. In this research one of the most renowned method called Burrows-Wheeler transform (BWT) has been used. This method is used to sort data reversibly. After this transformation, most of the repeated characters come together. The result of BWT has been passed through Run Length Encoder (RLE) module to get the Run characters, Run Length and Run Character frequencies. The Run characters and Run Character Frequencies have been passed through Move-to-Front (MTF) module to get MTF number and final MTF list. Then the MTF number and Run length, which are non-negative integers, have been sorted using counting sort module. This module sorts the numbers according to the Run characters. The result is non-negative integers which have been coded using interpolative coding method. This method is also known as Non-statistical and Non-parametric Coding. The data has been decompressed using the same modules as compression but in reverse order. It results lossless output. To implement vectorization big files are broken into chunks and passed through each module, creating new processes. So, it works in parallel. This method results speed boost exponentially with file size while keeping compression factor intact if not better.

Keywords: BWT, Integer Compression, Lossless Compression, Non-parametric coding, Non-statistical Coding, Run Length Encoding, Move to Front coding, Vectorization

1 Introduction

Computer memory, also known as a hierarchy of storage devices, ranges from slow to fast and inexpensive to expensive. Disk and tape are said to be slow and inexpensive whereas registers or CPU cache are considered as fast and expensive. Running an application is directly dependent on accessing to slower storage devices. Previously, when disks and tapes were considered slow, developers were focused on to optimize these devices only. Later, CPUs became so fast that accessing to main memory became limiting factor or bottleneck for many works. If the bandwidth requirement for accessing the main memory can be reduced, then the speed to perform a task can definitely be increased. Using data compression, the query performance can be improved (1).
Information theory is characterized as the study of efficient coding and its outcomes, in the form of transmitting speed and possibility of error (2). Information theory is the base on which compression algorithms have been built. In a sense, we've come from ancient computing with limited memory and bandwidth to modern mobile computing with limited memory and costly data plans. Since the limitation of bandwidth and large size of contents, data compression has become essential practice to perform to provide higher quality of service in digital world.
Since the first publication in 1994, researchers in the lossless compression field have been particularly interested in the Burrows Wheeler Transformation (BWT) (3). Besides the theoretical interest of BWT, highly practical lossless compression schemes can use it as a basis. BWT-based schemes usually compromise between the high compression rate of the prediction by partial matching (PPM) method with the pace of dictionary-based methods such as the variants Lempel-Ziv.
The compression process usually consists of two main parts: modeling and coding. Modeling usually produces the input for coding, together with the probability distribution, a set of (possibly transformed) items. The source data is transformed in many modelling methods into a series of smaller integers which can be encoded more compactly than the original items (4). In this research, the proposed method uses vectorization technique to compress/ decompress integer sequences. If the integers are small, the representation can be done using short codes. For the modelling purpose, broadcasting technique has been used. It makes modelling faster and computation less complex.
In this research, integer coding is proposed to perform through vectorization. BWT uses necessary input symbols as the sort key to sort the data uniquely. Similar data are grouped together creating a cluster. The data in same cluster contains same character a number of times and only few symbols can be there based on similar contexts (5).
This research is focused on implementing proper methods as post-transformation stage to achieve better compression as well as implementing proper preprocessing model to handle any kind of files. Also the main focus of this research is to obtain better compression and decompression speed by using non-parametric coding method. Since the output of post transformation

is non-negative integer, non-statistical compression technique has been used to compress the data. Also, it is expected to exploit the superscalar nature of modern processors and Single instruction multiple data (SIMD) instructions to achieve the goal (6). In previous work, Interpolative coding has been used to code the data. It gives better compression but a lot slower than other schemes. In this research it's been tried to obtain an improved trade-off between compression rate and time to encode/ decode using SIMD. This technique works on multiple chunks in parallel (7).
In summary, this paper makes the following contributions:

- Using broadcasting technique in clustering, to model large set of data in parallel
- Perform Vectorization of integer coding to achieve better compression in less time
- Handle non-text files as well as large files with same proficiency

This paper is organized as follows. Section 2 introduces Related work of Research. Section 3 describes Compression using Vectrorized processing. Section 4 explains the setup and results. This section also explains some key points of this research. Finally, we conclude this paper in Section 5.

2 Related Work

2.1 Burrows-Wheeler Transform

Bzip2 is one of the recent powerful software program for efficient data compression. It is one of the best general purpose compression tool for text. The base of this program is Burrows-Wheeler transform algorithm. The BWT algorithm was introduced by Burrows and Wheeler in 1994 (3).
Since the output contains many repeated characters, it can be compressed easily. The remarkable thing about this transform is that it generates a string which is easy to encode. It's better than sorting because in sorting the reverse process does not exist. But BWT allows the original document to be generated using the last column of data and index of original data.
The encoding algorithm changes the order of the symbols of source by manipulating the entire source sequence. The original sequence of symbols can be obtained by using decoding process. The permutation of all input sequence of symbols is taken during encoding which results a new sequence containing favorable features for compression.

2.2 Local-to-Global Transform

Compression is directly dependent on the context of the source. When it changes, the symbol distribution within the BW transformed data also changes. These changes can perform dramatic effect on the performance of the algorithm. Using entropy coders to react to these changes cannot be enough even when it is adaptive. For this reason, Local to Global Transform (LGT) is

employed in most methods (4). The purpose of this stage is to transform the local structure of the BWT output into global structure which can be considered more stable over the entire file.
In the original publication of BWT, Burrows and Wheeler suggested a recoding method as LGT. They used Move-to-Front (MTF) as LGT. This method converts a given input characters into integer denoting their last position and move it to the front. It is described in more detail in next section. Many authors believe that using this method is adding unnecessary complication. Some has suggested to get rid of it to obtain better compression but at slower rate. The major drawback of using MTF is that the contextual information is lost in the way. But other authors have mentioned that MTF can represent a better compromise between speed and compression rate.

2.3 Entropy Coding

Entropy coding is used as final stage in any BWT-based compression method. In this stage the actual compression is performed. Huffman coder was used in the original paper of Burrows and Wheeler as the final stage for encoding MTF numbers. Later arithmetic coder was used to improve the compression rate. In the recent research, Niemi A. and Teuhola J used interpolative coding to encode MTF numbers (9). They implemented run length encoding and MTF in this order and sorted their outputs based on run characters. It was then coded using interpolative coding. In the decompression process the integers were sorted using Move from Front module.

2.4 Integer Coding

The second and most important part of compression is coding. In this research the model generates non-negative integers. To compress and store integer can be costly to CPU. The main focus is on vectorization of integer coding algorithm. Some work has been done in this domain. we have tried some renowned methods to achieve expected results.

2.4.1 Variable byte and byte oriented coding

This coding is very popular. It is known under different names such as v-bytes, variable-byte, vbyte, VInt, varint and var-byte. It codes the data in the unit of bytes. The 7 lowest order bits of a byte is to store the original data and the eighth bit is used to denote whether next bit is used to denote the same integer. There are other variable byte encoding mechanisms like binary packing, the simple family and patched coding (7).

2.4.2 Semi-fixed Length Coding

When the upper bound of an integer that needs to be coded is not a power of two, Semi-Fixed length coding is used (8). It's also called “truncated binary coding” by Golomb. It is a prefix code which contains two code word lengths:

n and n-1 bits. There are many number of possible assignments of semi-fixed-length codewords to the n integers, but only the four shown in table 1 are used as it is easily computable:

Table 1 Semi Fixed Coding

Number	Low Short	Mid Short	High Short	Mid Long
0	100	0000	0000	100
1	111	0011	0011	0000
2	0000	0100	100	0011
3	0011	0111	111	0100
4	0011	100	0100	0111
5	0100	111	0111	111

3 Methodology

The compression starts with an input file. In previous works, only text file has been compressed. But in this research, the preprocessing algorithm handles the file in different format and convert it into appropriate format for further processing. Algorithm 1 shows the overall process.

Algorithm 1 BW Clustering Vectorization Coder

```
1: Get input file from user
2: Perform preprocessing
3: Tune the data to adapt Vectorization
4: BWT(preprocessedData) = bwtIndex + TransData
5: RLEncoder(TransData) = runChar + runLen + runCharFreq
6: MTF(runChar, runCharFreq) = mtfNum + finalMtfState
7: Sort(mtfNum, runChar, runLen, runCharFreq) = sortedMtfNum + sortedRunLen
8: b = encoder(bwtIndex)
9: b.append (encoder(finalMtfState))
10: b.append (encoder(runCharFreq))
11: b.append (encoder(sortedMtfNum))
12: b.append (ecoder(sortedRunLen))
```

The file content can be normal text file or binary file like images, spreadsheet, mp4 etc. These files can't be opened and treated the same. So, before any process starts the files are categorized into binary vs non-binary files. Also the files with different encoding than UTF-8 or UTF-16 are also treated as binary files. The preprocessing algorithm is shown in algorithm 2:

Algorithm 2 Preprocessing

```
1: Get input file from user
2: Check file size and split into chunks if necessary
3: Try to read input file normal way (UTF-8 encoding)
4: If step 3 fails, read file in binary format
5: Return binary index, data
```

The compression and decompression works in reverse fashion. For decompression, same modules are used but in reverse order. In this research for desorting the MTF numbers a reverse method has been used called Move-From-Front (9). This name is given according to its work. It moves the data from front to its original location using final mtf list.
Changing the context changes the symbol distribution in the BW-transformed data. These changes directly affect the encoder performance. Even the Encoder is adaptive, it may not react smooth enough to adapt such changes. To overcome this effect, a method called Local-to-Global Transform is employed after BWT. Each module is explained in detail in following sub-sections.

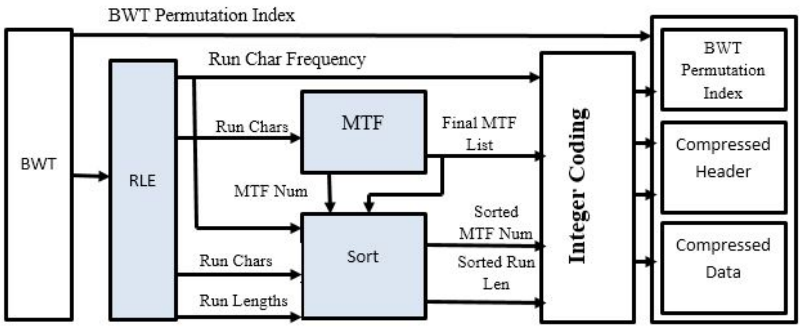


Fig. 1 Block Diagram of Compressor

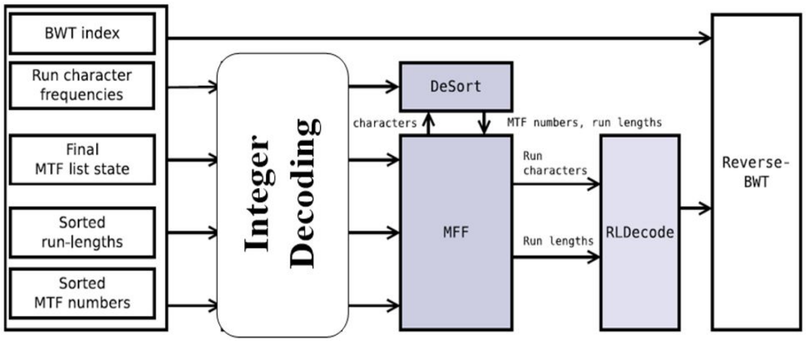


Fig. 2 Block Diagram of Decompressor

3.1 Vectorization of Input Files

In this research we have tried to compress and decompress big files using Vectorization method. This method lets us to exploit multiprocessing feature of computers manually.
This technique is similar to MapReduce method. Big files are broken into smaller chunks using splitter and each chunk is Mapped to parallel processes. After the process is finished, the Mapper reduces the chunks into a single file. It is used for both Compression and Decompression of bigger files.
So whenever a file bigger than 500Kb is used then it passes through Vectorization module and creates number of chunks. This also affects the symbol distribution of a file. If a file contains repeated characters and splitted into more than one file then compression won't be as good as if it is treated as one file. But the plus side is, as the file size increases the compression rate may go down but compression speed will be a lot faster than other traditional methods. It is shown in figures 3 and 4.

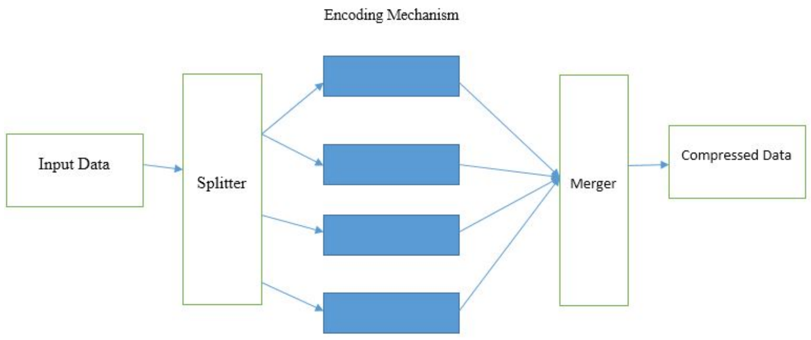


Fig. 3 Compression using Vectorization

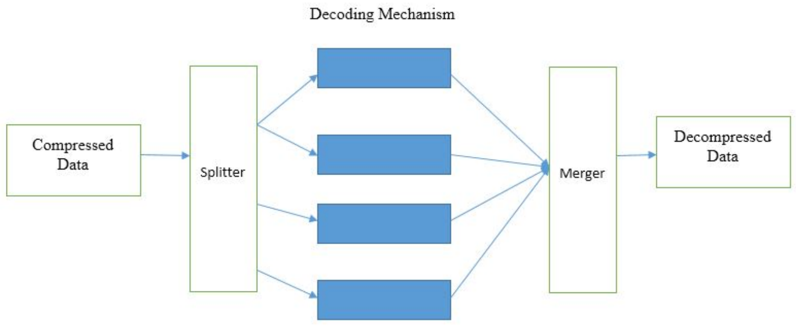
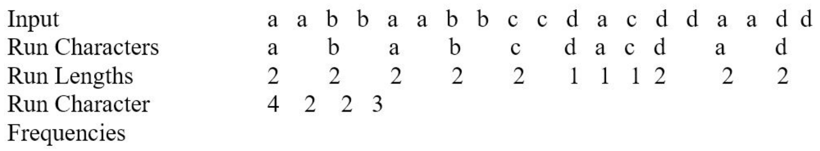


Fig. 4 Decompression using Vectorization

3.2 Run Length Encoder

The traditional way of run-length encoding has been used. The two-vector approach has been used along with it i.e Run char and Run length. Due to interpolative coder characteristics, run characters and their lengths are kept separately. The run characters are converted into MTF numbers by using MTF coder (10).
The RLE of our post-transformation method is shown in Figure 5. The run character frequencies are also computed and stored in alphabetical order. The MTF recoding uses this data to determine unique alphabets.



Input	a	a	b	b	a	a	b	b	c	c	d	a	c	d	d	a	a	d	d
Run Characters	a		b		a		b		c		d	a	c	d		a		d	
Run Lengths	2		2		2		2		2		1	1	1	2		2		2	
Run Character Frequencies	4	2	2	3															

Fig. 5 Example of run-length encoding of BWT output

3.3 Move-To-Front Coder

Each run character is converted to a number of distinct characters using MTF since its last presence. The two-vector method of RLE lets decrement each MTF number by one as it's described in above section. Figure 6 shows a descriptive example.
Run Character: a b a b c d a c d a d a
MTF list State:

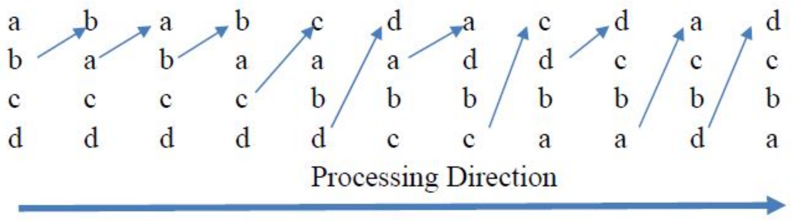


Fig. 6 MTF of the Run characters

3.4 Clustering by reversible sorting

MTF numbers are small and run lengths are long for common characters while it's an opposite case for uncommon characters. The triples of [MTF number, run character, run length] can be used to exploit this knowledge by clustering it. It is done by using run characters (4). Also, it must be remembered that the reversible operation of Clustering should be possible to recover the original

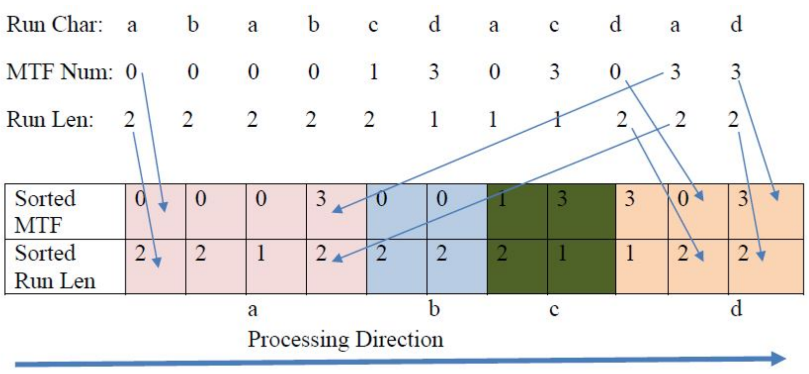

Run Char: a b a b c d a c d a d
MTF Num: 0 0 0 0 1 3 0 3 0 3 3
Run Len: 2 2 2 2 2 1 1 1 2 2 2
Sorted MTF
Sorted Run Len
0 0 0 3 0 0 1 3 3 0 3
2 2 1 2 2 2 2 1 1 2 2
a
b
c
d
Processing Direction

Fig. 7 Sorting

triples. Two arrays are created as output by the encoder. The one output is for MTF numbers and another output is for run lengths. In above section, we have said that the arrays are divided into bins. Figure 7 shows that the bins are implicit and the joined in alphabetic order. To show the next available position, a pointer is used. Each pointer initially points to the start of the respective bin. The triples, MTF number, run character and run length, are evaluated from left to right. The current run character is used to sort the current run length and MTF number. Finally, the pointers pointing to bin are increased by one, maintaining the stability.

3.5 Interpolative Coding

For encoding the MTF output, Interpolative coding has been used. As disucssed in section 2, it is a type of semi fixed coding. In previous research (9), different coding for leaf and nodes have been used. But in this research only one coding called Mid Short coding has been used. In this method the short codes are assigned to the middle integers and long ones are assigned to the numbers near the edges.
Let's say an array contains non-negative numbers.
A = 2, 5, 6, 8, 2, 10, 7, 3
Interpolative coding requires to put these numbers in tree structure as follow:
The root is coded using universal coding such as gamma coding (4) and the colored nodes are coded using semi fixed coder. As it can be seen the nodes contains the sum of its children. As the number of integers increases, so does the height of tree and the weight of each node. With increasing weight of nodes, the bits require to encode also increases. It is coded as parent child relationship using semi fixed coding.

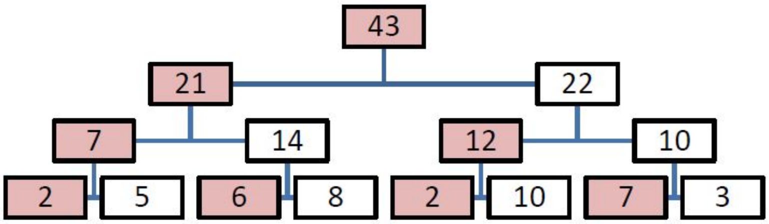


Fig. 8 Interpolative Coding

3.6 Move-From-Front Decoder

The decoder looks at the final state of MTF list and read the character at the front of the list. Then it applies the process called desorting to next number in the MTF list and Run length from the container or bin of that particular character. It reads the MTF number and then moves the character downward in MTF list by that number. This way it operates in reverse order than MTF (9).
Run character frequencies are used by the decoder to reverse the sorting. Since the decoder is operating in reverse order now, the pointer pointing to bins are now set to point to the rear of sort bins. MTF list is updated as well as bin pointer is decremented after desorting. This process continues as iterative process and shown in figure 9.

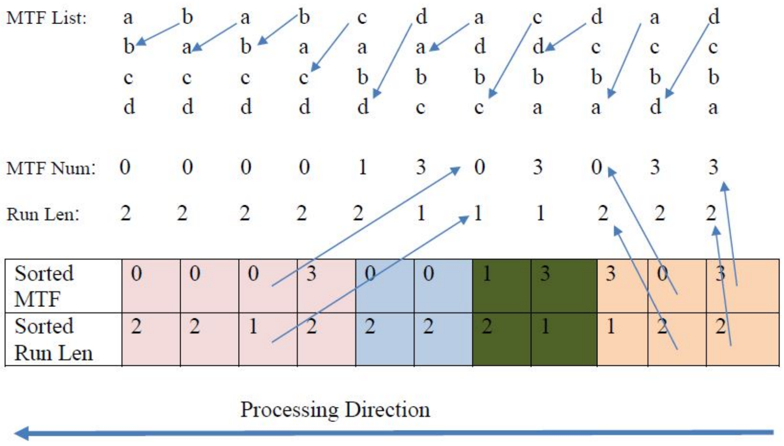

MTF List:
MTF Num: 0 0 0 0 1 3 0 3 0 3 3
Run Len: 2 2 2 2 2 1 1 1 2 2 2
Sorted MTF
Sorted Run Len
Processing Direction

Fig. 9 Move From Front

4 Results and Discussion

4.1 Experimental Setup

For testing purpose, Google colab has been used. The final program runs on local machine. The local machine has 1 TB Hard disk, 6 GB Ram and Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz Processor.
For proper evaluation, the standard files have been used. Some of them are: Calgary corpus, Canterbury Corpus, Artificial Corpus, Large Corpus, Misc Corpus and some other standard files. The table 2 contains the corpus names and their contents.

Table 2 Standard Corpus list

Corpus	Number of Files
Calgary	19
Canterbury	12
large	3
misc	1
Princeton	5

These files are grouped into 4 classes: Small, Medium, Big and Bigger.They are categorized according to their size range.

Table 3 File class and Size

Class	File size Range
Small	Less than 500 Kb
Medium	500 Kb to 1 Mb
Big	1 Mb to 5 Mb
Bigger	More than 5 Mb

4.2 Read/ Write structure of files

The input file has been preprocessed using 2. The algorithm results a bit saying whether the input file is a normal text file or not. This bit is called binary flag. Then the file size is read to determine if splitting the file is required or not. So another flag bit called split flag has been used. If file is split then each chunk is passed through Encoding mechanism to process it. Otherwise, a single file is passed. After preprocessing is done, then it's been passed through BWT module which results the transformed file.
This module does not perform the compression but it helps other modules to compress data effectively. The index has been passed to final result as it is.

First, it is converted into binary format along with the number of bits required to represent it. So,

$$BwtHeader = Binary[index + length(indexinbinary]$$

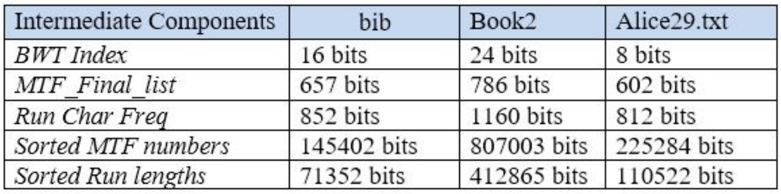
Then, the Transformed file has been passed through RLE module. This module produces three intermediate files: Run chars, Run length, Run char frequency. The Run chars are passed through MTF module which provides MTF list and MTF Final State. The Run chars, MTF list and Run char frequency are passed through sorting module. This module sorts the MTF list and Run chars to provide effective clusters for compression. The Run char frequency and MTF Final State are passed through encoder to generate Compression Header. The compression data is made up of sorted MTF list and sorted Run length.
The encoder encodes every file in this format:

$$InterEncodeData = EncodedData + leafLen + leafBit + root$$

Since Encoder module does not know where to stop, the length of leaves are also encoded along with the number of bits needed to represent it. The leaf bit has been used to dynamically code the length of leaves instead of fixed binary representation of leaves. The leaf bit is of 2 bit length which provides the information of how many bits has been used to represent the length of the leaf node. Since the output of Encoder module is self-explanatory, combining the output of other intermediate is not a problem. The decompression process starts in reverse order. First binary flag and split flag is taken care of then the BWT index is taken care of using its leaf bit. Then the remaining file is passed through Decoder module to list the intermediate files recursively. After the intermediate files are generated they are passed through their respective decoder modules to generate original file.

4.3 Structure of the output

The compressed file consists of compressed data and some extra information. The main components are: the BWT permutation index, compressed header containing mtf final list and run character frequency, and the last component is compressed data containing sorted mtf list and sorted run char length. All these components, except BWT permutation index is encoded with semi fixed length encoder. The BWT permutation index is coded using byte coder which uses a fixed byte to code. The leaf flag has been used to denote how many bytes have been used for BWT permutation index. Since, the encoder is non-statistical and non-parametric, the decoder decompresses each component separately before starting MFF process. Only then Run length decoder and BWT restore module runs. For most of the files the compressed header is less than 1% of the total size. So, the overhead of the post BWT stage seems reasonable. The data in figure 10 validates it.

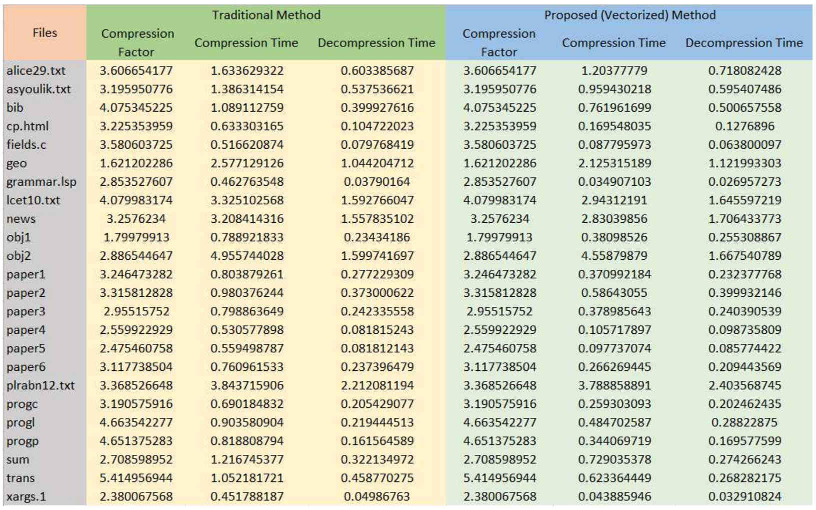


Intermediate Components	bib	Book2	Alice29.txt
BWT Index	16 bits	24 bits	8 bits
MTF_Final_list	657 bits	786 bits	602 bits
Run Char Freq	852 bits	1160 bits	812 bits
Sorted MTF numbers	145402 bits	807003 bits	225284 bits
Sorted Run lengths	71352 bits	412865 bits	110522 bits

Fig. 10 Component sizes of example output

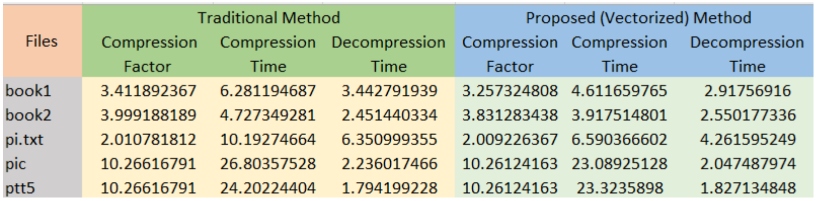
4.4 Experimental Data

For testing and Validation purpose, standard corpus have been used. The comparison is between previous related works (termed as Traditional method) and proposed method termed as Vectorized Method. Compression Factor and Compression Speed for different class of files are shown in below figures.



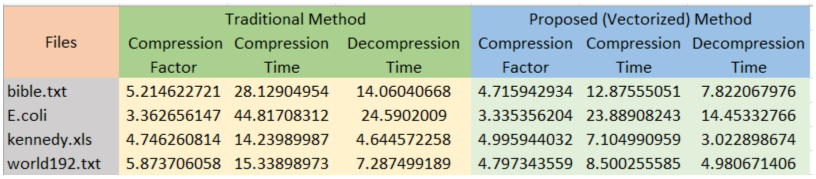
Files	Traditional Method			Proposed (Vectorized) Method		
	Compression Factor	Compression Time	Decompression Time	Compression Factor	Compression Time	Decompression Time
alice29.txt	3.606654177	1.633629322	0.603385687	3.606654177	1.20377779	0.718082428
asyoulik.txt	3.195950776	1.386314154	0.537536621	3.195950776	0.959430218	0.595407486
bib	4.075345225	1.089112759	0.399927616	4.075345225	0.761961699	0.500657558
cp.html	3.225353959	0.633303165	0.104722023	3.225353959	0.169548035	0.1276896
fields.c	3.580603725	0.516620874	0.079768419	3.580603725	0.087795973	0.063800097
geo	1.621202286	2.577129126	1.044204712	1.621202286	2.125315189	1.121993303
grammar.lsp	2.853527607	0.462763548	0.03790164	2.853527607	0.034907103	0.026957273
lcet10.txt	4.079983174	3.325102568	1.592766047	4.079983174	2.94312191	1.645597219
news	3.2576234	3.208414316	1.557835102	3.2576234	2.83039856	1.706433773
obj1	1.79979913	0.788921833	0.23434186	1.79979913	0.38098526	0.255308867
obj2	2.886544647	4.955744028	1.599741697	2.886544647	4.55879879	1.667540789
paper1	3.246473282	0.803879261	0.277229309	3.246473282	0.370992184	0.232377768
paper2	3.315812828	0.980376244	0.373000622	3.315812828	0.58643055	0.399932146
paper3	2.95515752	0.798863649	0.242335558	2.95515752	0.378985643	0.240390539
paper4	2.559922929	0.530577898	0.081815243	2.559922929	0.105717897	0.098735809
paper5	2.475460758	0.559498787	0.081812143	2.475460758	0.097737074	0.085774422
paper6	3.117738504	0.760961533	0.237396479	3.117738504	0.266269445	0.209443569
plrabn12.txt	3.368526648	3.843715906	2.212081194	3.368526648	3.788858891	2.403568745
progc	3.190575916	0.690184832	0.205429077	3.190575916	0.259303093	0.202462435
progl	4.663542277	0.903580904	0.219444513	4.663542277	0.484702587	0.28822875
progp	4.651375283	0.818808794	0.161564589	4.651375283	0.344069719	0.169577599
sum	2.708598952	1.216745377	0.322134972	2.708598952	0.729035378	0.274266243
trans	5.414956944	1.052181721	0.458770275	5.414956944	0.623364449	0.268282175
xargs.1	2.380067568	0.451788187	0.04986763	2.380067568	0.043885946	0.032910824

Fig. 11 Result of Small Class Files (Size less than 500 Kb)



Files	Traditional Method			Proposed (Vectorized) Method		
	Compression Factor	Compression Time	Decompression Time	Compression Factor	Compression Time	Decompression Time
book1	3.411892367	6.281194687	3.442791939	3.257324808	4.611659765	2.91756916
book2	3.999188189	4.727349281	2.451440334	3.831283438	3.917514801	2.550177336
pi.txt	2.010781812	10.19274664	6.350999355	2.009226367	6.590366602	4.261595249
pic	10.26616791	26.80357528	2.236017466	10.26124163	23.08925128	2.047487974
ptt5	10.26616791	24.20224404	1.794199228	10.26124163	23.3235898	1.827134848

Fig. 12 Result of Medium Class Files (Size between 500 Kb to 1 Mb)



Files	Traditional Method			Proposed (Vectorized) Method		
	Compression Factor	Compression Time	Decompression Time	Compression Factor	Compression Time	Decompression Time
bible.txt	5.214622721	28.12904954	14.06040668	4.715942934	12.87555051	7.822067976
E.coli	3.362656147	44.81708312	24.5902009	3.335356204	23.88908243	14.45332766
kennedy.xls	4.746260814	14.23989987	4.644572258	4.995944032	7.104990959	3.022898674
world192.txt	5.873706058	15.33898973	7.287499189	4.797343559	8.500255585	4.980671406

Fig. 13 Result of Big Class Files (Size between 1 Mb to 5 Mb)

4.4.1 Compression Factor

As we can see in figure 14 for small class files, the compression factor does not differ in two methods. The reason behind this is because the files are smaller than the chunk size defined. Hence both methods result same compression factor.

The medium and large class files are bigger than 500 kb which is bigger than our default chunk size. From the compression factor graph 15 and 16, it can be seen that files have different compression factor between two methods. For some files compression is better in traditional method. This happens when the content of the file is repetitive. When it is processed together it can result better compression but when it's divided into smaller chunks, the repetition of characters also breaks. So it results worse compression Factor.

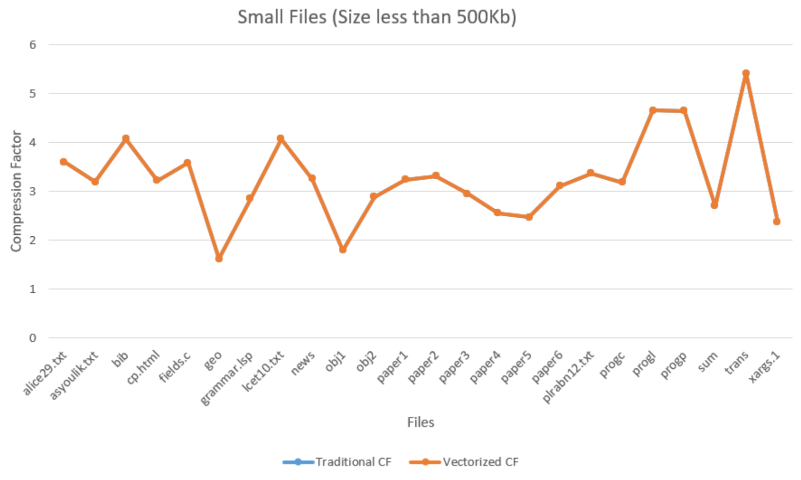


Fig. 14 Compression Factor of Small Class Files

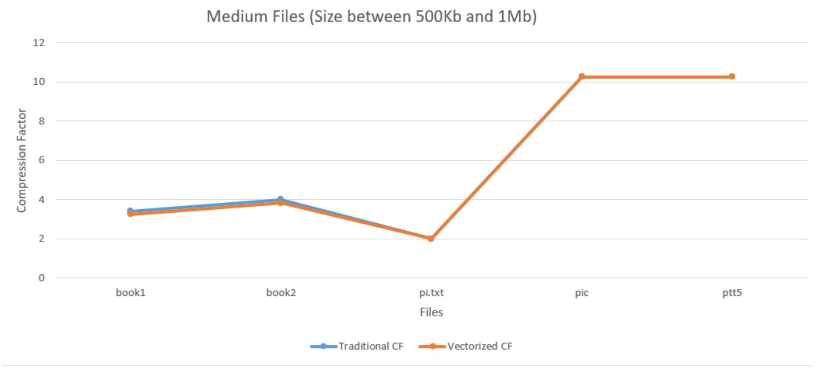


Fig. 15 Compression Factor of Medium Class Files

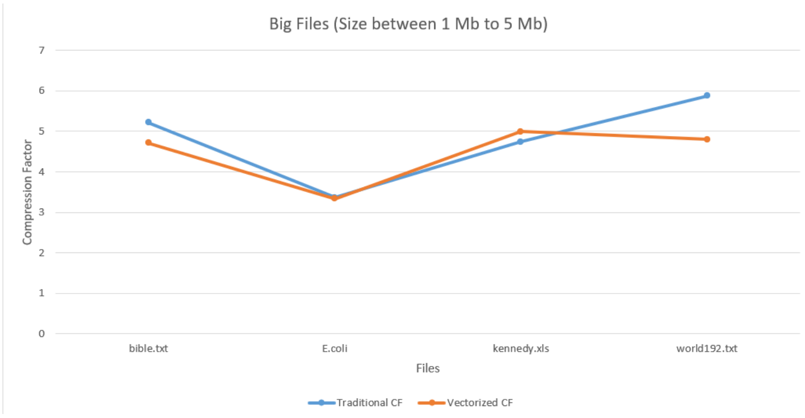


Fig. 16 Compression Factor of Big Class Files

4.4.2 Compression Speed

The time taken to compress different class of files have been analyzed. As the file size has increased the difference between time taken by Traditional method and Vectorized method has also increased. Vectorized method results better speed as we move from small class to big class files. Figures 17, 18, 19 shows Compression time between traditional and vectorized method.
Figure 20 shows compression time vs Decompression time of Vectorized method. We can see that Decompression speed is a lot faster than Compression Speed.

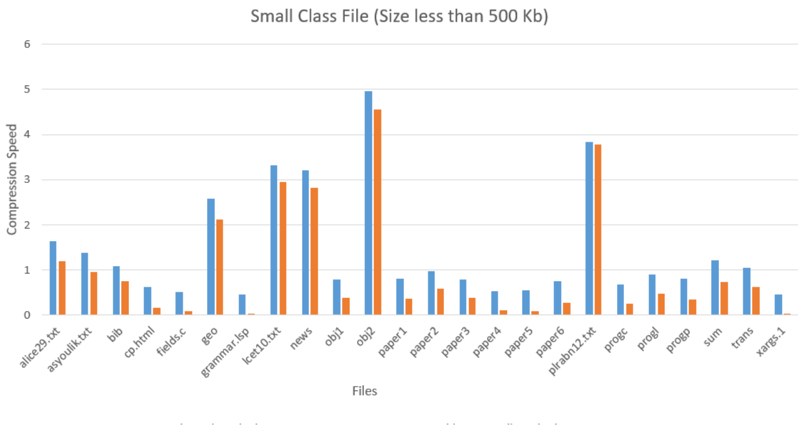


Fig. 17 Compression Speed of Small Class files

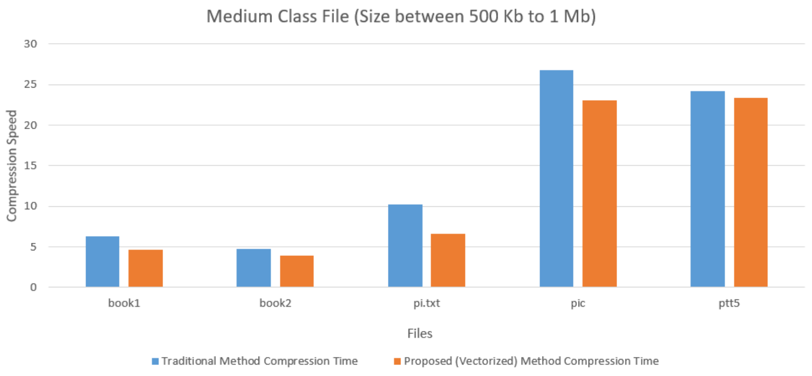


Fig. 18 Compression Speed of Medium Class files

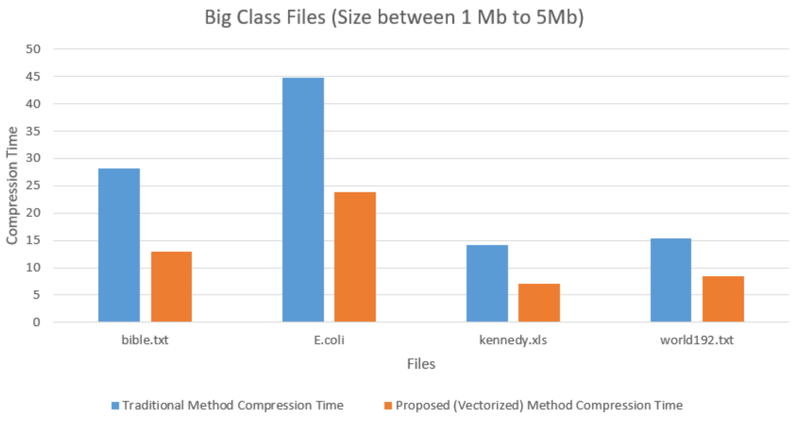


Fig. 19 Compression Speed of Big Class files

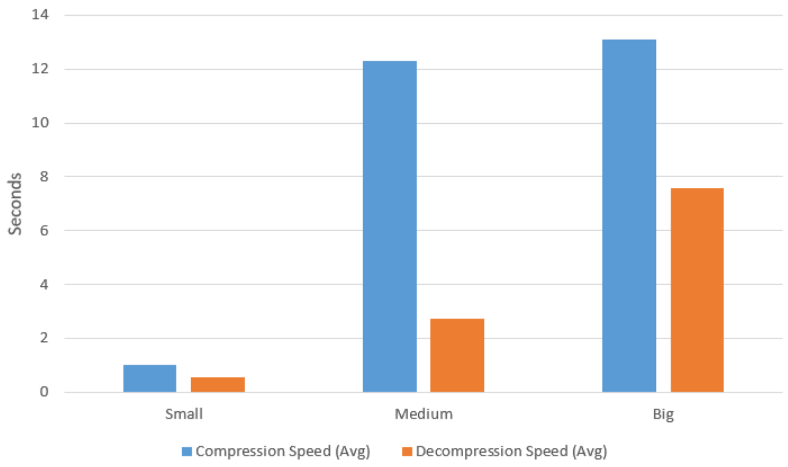


Fig. 20 Compression Speed compared to Decompression Time

5 Conclusion and Future Work

In this research, non-statistical coding method has been used to perform compression. A better implementation of BWT method reduced time significantly. The traditional method could not be used to compute because of memory overflow. But by using Suffix Array it's only a matter of second to get preprocessed data. Then, the only problem was traversing through the binary tree during coding and decoding. Better implementation of this coding/ decoding method could result faster and better result. The vectorization of integer coding has been implemented. Handling bigger files as well as non-text file, it gave better result than many other non-statistical algorithms available these days.
Up to now, most of the compression algorithm works serially. But the proposed method works in parallel. Hence the compression/ Decompression speed increases. Also it became easier to work with bigger files.
For future work, the partition of files can be handled with more care. The work is based on Integer compression, so the compression factor is not good for all files. The compression is slightly worse than statistical methods. But if the partition of chunks can be handled carefully, this drawback can be removed. Also, different encoding technique can be tested to improve speed and compression factors.

Declarations

Acknowledgements. Not applicable.

Author Contributions. AKY:Envisioning the idea, defining the research problem and contributing to the solution of the problem, and editing the paper. SPP: Contributing to the solution of the problem and doing the implementation of the paper and editing the paper. All the authors of this paper have collaborated and worked toward developing the paper and made significant contributions to the paper.

Funding. Not applicable

Conflict of interest. The authors declare that they have no conflict of interest.

Data availability. Yes, all related materials concerning this paper are available with us and can be provided upon request.

Code availability. Yes, the code is available with us.

References

[1] Timothy Bell, Ian H Witten, and John G Cleary. Modeling for text compression. ACM Computing Surveys (CSUR), 21(4):557–591, 1989.

[2] Jukka Teuhola. Tournament coding of integer sequences. The Computer Journal, 52(3):368–377, 2009.

[3] Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. In Digital SRC Research Report. Citeseer, 1994.

[4] Alistair Moffat and Lang Stuiver. Exploiting clustering in inverted file compression. In Proceedings of Data Compression Conference-DCC'96, pages 82–91. IEEE, 1996.

[5] Peter Fenwick. Burrows wheeler compression. Lossless Compression Handbook,pages 169–193, 2003.

[6] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. Software: Practice and Experience, 45(1):1–29, 2015.

[7] Fabrizio Silvestri and Rossano Venturini. Vsencoding: efficient coding and fast decoding of integer lists via dynamic programming. InProceedings of the 19th ACM international conference on Information and knowledge management,pages 1219–1228, 2010.

[8] Alistair Moffat and Lang Stuiver. Binary interpolative coding for effective index compression.Information Retrieval, 3(1):25–47, 2000

[9] Arto Niemi and Jukka Teuhola. Burrows-wheeler post-transformation with effective clustering and interpolative coding.Software: Practice and Experience, 50(9):1858–1874, 2020.

[10] Borut Zalik, Domen Mongus, Niko Luka˘c, and Krista Rizman˘Zalik. Efficient chain code compression with interpolative coding.Information Sciences,439:39–49, 2018.