

CALIFORNIA STATE UNIVERSITY
NORTHRIDGE

COMP 620 - COMPUTER SYSTEM ARCHITECTURE

(21912 - FA2022)

**Department of Computer Engineering
and
Computer Science**



Instructor: David Freedman

LZW - COMPRESSION

**OMKAR LUBAL - 203035397
DEVAMSH KONDRAGUNTA - 203125890**

SECTION 1

INTRODUCTION

We look into the idea of leveraging several processors to speed up Lempel-Ziv-Welch scheme encoding and decoding. It is recommended that the processors be rearranged using a full binary tree, and it is demonstrated how LZW and LZW can be modified to benefit from these parallel designs. The design is then expanded to include higher order trees. Experimentation reveals an increase in time over the sequential technique and an improvement in compression over the common way of parallelization method.

Approaches for compression are frequently divided into static and dynamic methods. The static approaches presuppose that the file that needs to be compressed was created using a specific model that was fixed in advance and was known to both the compressor and decompressor. The probability distribution of the various characters or, more generally, of certain variable-length substrings that appear in the file, together with a method to parse the file into a precisely predetermined sequence of such elements, could serve as the basis for the model.

Applying a statistical encoding function, such as Huffman or arithmetic coding, will then yield the encoded file. The compression procedure can only be carried out in a second pass because information about the model is either presumed to be known or may be acquired in a first pass through the file.

COMPRESSION ALGORITHMS

Utilizing fewer bits than the original representation to encrypt information. Uncompressed data can occupy a large amount of space, which is bad for hard drive space limitations and slow internet download speeds. While hardware improves and becomes more affordable, technology also advances thanks to data compression methods. But a lot of widely used compression techniques are adaptive. It is not presumed that the underlying model is known; rather, it is found when the file is processed sequentially.

For compressor and decompressor to operate in synchrony without needing to send the model itself, the encoding and decoding of the i th element is based on the distribution of the $L-1$ preceding ones. The Lempel-Ziv Welch (LZW) methods and their variations are examples of adaptive methods, but there are also adaptable versions of the Huffman and arithmetic coding.

To decrease the encoding and decoding times, we want to investigate the use of several CPUs. For static Huffman coding, this has been done in, with an emphasis on the decoding procedure

in particular. The current project looks into how LZW coding could benefit from parallel processing.

With n processors, algorithms for the PRAM model can complete this task in $O(m + \log(n))$ time, where m is the length of the longest dictionary entry and here n is the length of the string given as input. We take into account what is arguably the greatest real-world example of massively parallel computation: a linear array of processors with each processor solely connected to its left and right neighbors.

We offer a method that, for every integer $k \geq 1$, runs in time $O(km + m \log(m))$ with $n/(km)$ processors and is certain to be within a factor of $(k+1)/k$ of optimal. We also offer research showing that performance can be even more effective in practice.

Our research focuses on LZW methods. Each item in the sequence of the encoded file in LZW is a pointer or a single character, of the type (off, len) which takes the place of a string of length 'len' that was earlier in the file, 'off' characters.

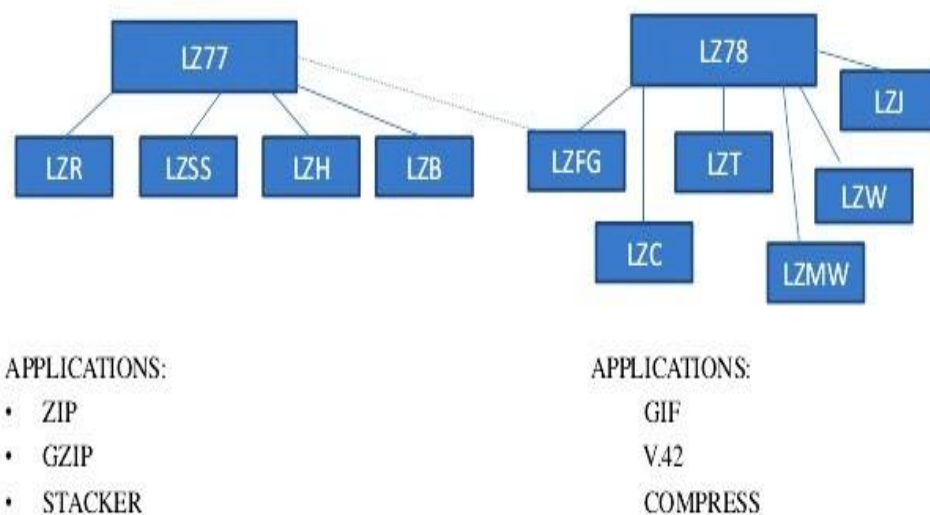


Fig.: Lempel Ziv Algorithm Family

LEMPERL-ZIV WELCH COMPRESSION

Abraham Lempel, Jacob Ziv, and Terry Welch developed the table-based lookup technique known as LZW compression to compress a file into a smaller file. The TIFF image format and the

GIF image format are two frequently used file formats that employ LZW compression. A specific LZW compression technique makes an entry in a table (also termed a "dictionary" or "codebook") for each input sequence of bits of a defined length (for instance, 12 bits), which has a shorter code and the pattern itself.

The Lempel-Ziv Welch algorithm is one of many algorithms used for compression. It is typically used to compress certain image files, unix's 'compress' command, among other uses. Text files can also be compressed using LZW compression.

The main idea relies on recurring patterns to save data space. In addition to PDF and TIFF, this method is commonly utilized in GIF. Since it is lossless, no data is lost during compression.

A dictionary of strings (symbol sequences) found in the input sequence is created by the LZW algorithm. A string will be encoded with the index of the related dictionary entry whenever it recurs.

Normally, 8 binary bits are used to store each character, giving the data room for up to 256 different symbols. It is typically used to compress certain image files, unix 'compress' command, among other uses.

Any pattern that has been read previously causes the shorter code to be substituted as input is processed, effectively compressing the overall amount of input into a smaller quantity. The look-up table of codes is part of the compressed file created by the LZW method unlike older methods known as LZ77 and LZ78. The algorithm is used by the decoding program to process the encoded input string, which enables it to construct the table as it decompresses the file.

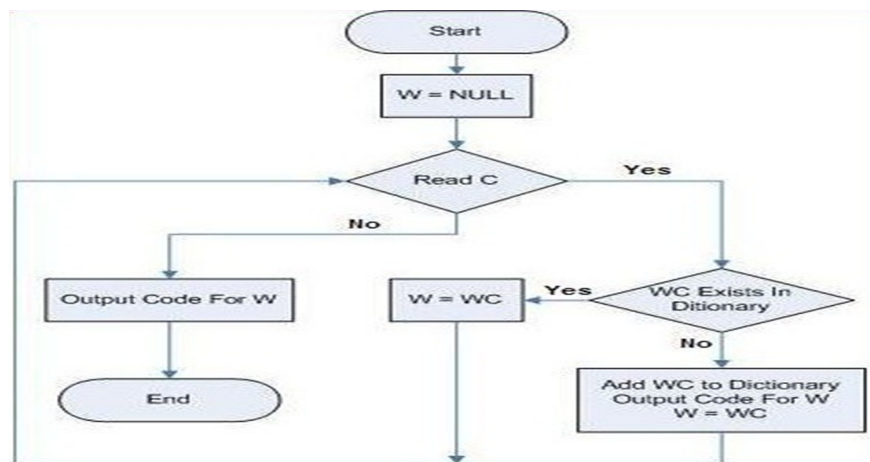


Fig.: LZW Algorithm Flow Chart

EXAMPLE

Input String: /WED/WE/WEE/WEB/WET

Character Input	Code Output	New code value	New String
/W	/	27	/W
E	W	28	WE
D	E	29	ED
/	D	30	D/
WE	27	31	/WE
/	E	32	E/
WEE	31	33	/WEE
/W	32	34	E/W
EB	27	35	WEB
/	B	36	B/
WET	31	37	/WET
EOF	T		

Fig.: LZW Data Compression Process

- The Idea: rely on recurring patterns to save data space.
- Since it is lossless, no data is lost during compression.
- Many communication and storage systems use the Lempel Ziv Welch (LZW) algorithm, a significant dictionary-based data compression technique.

COST & PERFORMANCE (SINGLE THREAD JVM)

Processor: 11th gen Intel(R) Core(TM) i5-1135G7 @ 2.4 GHz

Available processors (cores): 1

Free memory (MB): 250 MB

Maximum memory (GB): 3 GB

Total memory available to JVM (MB): 252 MB

SECTION 2

SEQUENTIAL APPROACH

An algorithm that is run sequentially—once through, from beginning to end, without any processing or execution in between—is known as a serial algorithm and is opposed to concurrently or parallel approach.

It begins adding symbol pairs to the word dictionary. Replace any pairs that have already been entered into the dictionary with the new value when we get to those pairs. We only need 9 bits for this short string, which gives us room for up to 512 different symbols.

This approach aims to increase the character set of this library to 9–12 bits. Combinations of symbols that have previously appeared in the string make up the new, distinctive symbols. sometimes fails to compress well, especially when dealing with short, erratic strings. The approach can both compress and uncompress data, therefore it is useful for compressing redundant data and does not require saving the new dictionary along with the data.

Thus, decoding such a file is quite simple. However, in order to discover the longest repeating sequences during encoding Hash tables and binary trees, two complex data structures have been proposed. The encoded file in LZW is made up of a series of pointers to a dictionary, each of which replaces a string from the input file that has previously appeared and been added to the dictionary. [2]

As a result, the dictionary must be created identically by the encoder and decoder. Putting a lower limitation on the size N/n of each block, however, essentially places an upper limit on the number of processors n which may be used for a given file of size N , thus we might not fully utilize all the available computing capacity.

However, it should be noted that equi-sized blocks cannot be assumed for both encoding and decoding at the same time. If blocks of a fixed size are used for encoding, the compressed blocks that result have varying lengths.

Therefore, one must either do a priori compression on blocks of variable sizes, resulting in compressed blocks that are all around the same size, or one must have a compressed file that should also contain a vector of indices to each processor's beginning point, adding extra storage overhead.

Then, one must add a few bits of padding to each block to ensure that they are all exactly the same size and to achieve byte alignment, but in this situation, the compression loss caused by the padding is typically insignificant. We require a form of "time stamp" that shows the stage at which an element was added to a Table. Only the indices of the final element for each block need to be recorded if the elements are sequentially kept in these tables. However, as LZW implementations typically utilize hashing to maintain the tables, it is impossible to infer information from its physical placement and each element must be designated specifically. The simplest method is to keep the index I of the block that led to the addition of P together with each string P . For each entry, $\log_2 n$ bits would be needed.

On the other hand, all the processors might be able to interact, causing delays that make this variation take as long as a sequential algorithm to complete. The proposed trade-off is based on a hierarchical network of connections between the processors, with each processor depending on no more than $\log n$ others.

The task can be completed in $\log n$ sequential phases by n processors working in parallel. The compression ratio will decline, but the loss will be less than when each of the n CPUs operates independently.

ASCII TABLE

ASCII is the name of the character encoding standard used for text files on computers and other devices (American Standard Code for Information Interchange). ASCII is a subset of Unicode and has 128 symbols in its character set. These symbols include capital and lowercase letters, numbers, punctuation marks, special characters, and control characters. Each character in the character set has a decimal value between 0 and 127, as well as comparable Hexadecimal and Octal values.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

128	Ç	144	È	160	Á	176	ð	192	Ł	208	ł	224	α	240	≡
129	à	145	é	161	â	177	í	193	ł	209	ŧ	225	ß	241	±
130	ê	146	æ	162	ó	178	ñ	194	ŧ	210	ŧ	226	Γ	242	≥
131	â	147	ô	163	û	179	ı	195	ı	211	ı	227	π	243	≤
132	ä	148	ö	164	ñ	180	ı	196	ı	212	ı	228	Σ	244	ƒ
133	å	149	õ	165	Ñ	181	ı	197	ı	213	ı	229	σ	245	Ƶ
134	ä	150	ü	166	ª	182	ı	198	ı	214	ı	230	μ	246	+
135	ç	151	ù	167	º	183	ı	199	ı	215	ı	231	τ	247	≈
136	è	152	ÿ	168	¿	184	ı	200	ı	216	ı	232	Φ	248	°
137	é	153	Û	169	ı	185	ı	201	ı	217	ı	233	⊙	249	·
138	ê	154	Ü	170	ı	186	ı	202	ı	218	ı	234	Ω	250	·
139	ı	155	ı	171	½	187	ı	203	ı	219	ı	235	δ	251	√
140	ı	156	ı	172	¾	188	ı	204	ı	220	ı	236	∞	252	∞
141	ı	157	ı	173	ı	189	ı	205	ı	221	ı	237	φ	253	²
142	Ä	158	ı	174	«	190	ı	206	ı	222	ı	238	ε	254	■
143	Å	159	ı	175	»	191	ı	207	ı	223	ı	239	ı	255	ı

Fig.: ASCII Code

SEQUENTIAL APPROACH IMPLEMENTATION IN JAVA

```

List<Integer> encode(String s) {
    Map <String, Integer> dictionary = new HashMap<>();

    for (int i = 0; i < dictSize; i++)
        dictionary.put(String.valueOf((char) i), i);

    String prevChars = "";
    List<Integer> result = new ArrayList<>();
    for (char character : s.toCharArray()) {
        String charsToAdd = prevChars + character;
        if (dictionary.containsKey(charsToAdd)) {
            // already have an entry
            // mup
            prevChars = charsToAdd;
        } else {
            result.add(dictionary.get(prevChars));
            dictionary.put(charsToAdd, dictSize++);
            prevChars = String.valueOf(character);
        }
    }

    if (!prevChars.isEmpty()) {
        result.add(dictionary.getOrDefault(prevChars, dictSize++));
    }
    return result;
}

```


MULTITHREADING

This is an ability of a central processing unit (CPU) to provide a number of threads of operation concurrently is known as multithreading, and it is supported by the operating system. The multithreading paradigm has gained prominence as efforts to further leverage instruction-level parallelism have languished since the late 1990s. This allowed for the reemergence of throughput computing from the more specialized domain of transaction processing.

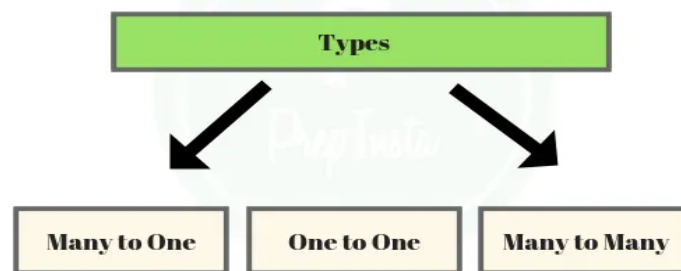


Fig: Types of Multithreading Models

CONCURRENCY

Concurrent execution is possible only when a multithreaded process takes place in a single processor because the processor can switch execution resources between threads.

PARALLELISM

Each thread in a shared memory multiprocessor environment multithreaded operation can execute concurrently on a different processor, resulting in parallel processing.

PARALLEL APPROACH

Asynchronous dictionary that contains the most highly used English words. Breakup the input into smaller chunks and process them parallelly. Moreover, the Parallel LZW has relatively high memory requirements which dominate the costs of a hardware implementation. This work proposes a large dictionary structure and word partitioning technique that allows to process workloads in a parallel manner.

- Synchronized dictionary that contains the most highly used English words.
- Breakup the input into smaller chunks and process them parallelly.

Partitioning the input file of size N into m blocks of size N/m and allocating each block to one of the m available processors is the fundamental concept behind achieving parallel compression. The encoding is then simple for static methods, but since the compressed file is divided into equal-sized blocks for the decoding, there may be a synchronization issue at the block boundaries.

Our new parallel coding approach is based on a trade-off between time and compression effectiveness that is proportional to the level of parallelization. One extreme is full parallelization, where each of the n processors functions independently. If the block sizes are tiny, this may significantly limit the compression as well as time gain.

Alternatively, one could design a parallel decoding process for static Huffman codes that allows each processor to decode one block but allow the data to overflow into one or more subsequent blocks until synchronization is achieved by taking advantage of the tendency of static Huffman codes to quickly resynchronize after an error.[1]

The additional issue for dynamic techniques is that the encoding and decoding of components in the i th block may be dependent on elements in some earlier blocks. It would still be substantially comparable to a sequential model even if one had a CREW architecture in which all the processors shared some common memory area that could be accessed in parallel.[1]

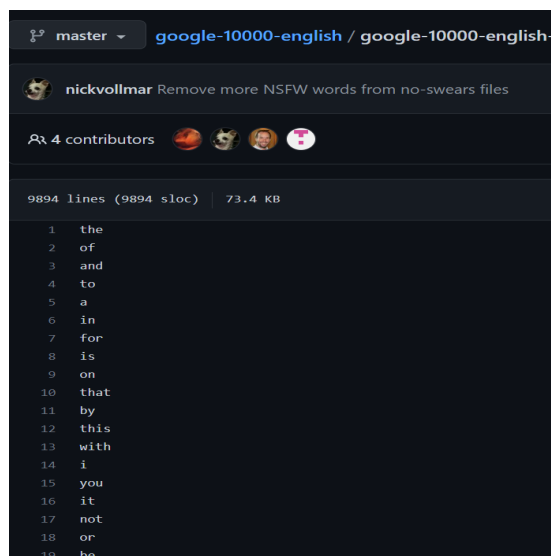
Despite the aforementioned issue, allowing one processor to operate independently of the others is the simplest way to accomplish parallelization. As a result, the file is divided into n blocks that are independently encoded and decoded on each CPU. This solution might even be suggested if the block size is large enough. Empirical testing has shown that expanding this history beyond a certain point quickly approaches zero, and most LZ systems have a limit on how much history is taken into account for the present item. Since each CPU must "learn" the key characteristics of the file, parallelization would result in a little decline in compression performance at block boundaries. However, this loss is frequently overlooked because it may enable a factor of n reduction in processing time.

For example, in an information retrieval system built on a sizable static database, compression is only performed once, therefore, the parallelization's increased speed might not matter. But whereas decompression of specific parts is necessary for each query to be processed, increasing the importance of parallel decoding. LZW creates an ever-expanding Table, but it is not necessary to broadcast it because the decoder reconstructs it synchronously. The set of single characters that make up the text, which is frequently taken to be ASCII, is initialized in the table.

For LZW methods, encoding and decoding are more closely related than for Huffman coding, whether or not the possibility of having several processors at the time of decoding was understood at the time of encoding, where parallel decoding may be utilized. As a result, we also need to deal with the parallel encoding strategy. We also assume that both activities can use the same number of processors.

According to the parallel LZW encoding, which refers to the characters in the input block as belonging to a vector which is stored in the memory of a processor and is also accessed by other blocks, each element stored in the Table requires an identifier indicating the block from which it has been generated. As a result, elements in the Table have the format (string, identifier). A series of pointers that represent the indices of the encoded elements in the Table constitute the LZW encoding's output. These pointers in our situation take the form of (index, identifier). The representation of the index, which addresses a smaller range and saves the extra bits needed for identification, maintains compression efficiency. [1]

For the sake of simplicity, we will skip over the specifics of handling incremental index encoding and overflow situations when the Table fills up. The same as for the serial LZW can be used. It is assumed that the input to the parallel LZW decode procedure is a sequence of components of the form (index, identifier). LZ compression algorithms take advantage of the fact that some strings have a tendency to recur soon after their first appearance, and that connecting blocks that are not adjacent perturb this locality of reference.



Rank	Word	Rank	Word	Rank	Word	Rank	Word	Rank	Word
1	the	21	this	41	so	61	people	81	back
2	be	22	but	42	up	62	into	82	after
3	to	23	his	43	out	63	year	83	use
4	of	24	by	44	if	64	your	84	two
5	and	25	from	45	about	65	good	85	how
6	a	26	they	46	who	66	some	86	our
7	in	27	we	47	get	67	could	87	work
8	that	28	say	48	which	68	them	88	first
9	have	29	her	49	go	69	see	89	well
10	i	30	she	50	me	70	other	90	way
11	it	31	or	51	when	71	than	91	even
12	for	32	an	52	make	72	then	92	new
13	not	33	will	53	can	73	now	93	want
14	on	34	my	54	like	74	look	94	because
15	with	35	one	55	time	75	only	95	any
16	he	36	all	56	no	76	come	96	these
17	as	37	would	57	just	77	its	97	give
18	you	38	there	58	him	78	over	98	day
19	do	39	their	59	know	79	think	99	most
20	at	40	what	60	take	80	also	100	us

Fig.: Github data for asynchronized dictionary

According to an examination of Google's Trillion Word Corpus using n-gram frequency, this repository includes a list of the 10,000 most frequent English terms in order of frequency. Such words having more frequency can be replaced with a smaller number (mostly a number directly after ASCII number limit) and lesser occurring words can take bigger numbers. Thus, allowing us a better compression ratio for this parallel approach.

This dictionary will then be asynchronously accessed by N processors who will start substituting words in their independent part of sentence. This approach allows us to achieve a faster compression speed and since the dictionary is only being read the problem of synchronization between N processors vanishes.

We will be using Google's Trillion Word Corpus to build our asynchronous dictionary which allows us to substitute most commonly used English words and compress our string.

Dictionary Data -> <https://github.com/first20hours/google-10000-english>

PARALLEL APPROACH IMPLEMENTATION IN JAVA

```
List<String> encode(String s) {
    dictionary = new HashMap<>();
    for (int i = 0; i < dictSize; i++)
        dictionary.put(String.valueOf((char) i), Integer.toString(i));

    populateDictionaryWithExtWords(dictionary);
    List<String> result = new ArrayList<>();
    int n = s.length();
    // use 4 thread
    CompletableFuture thread1 = CompletableFuture.runAsync(() -> {
        result.addAll(encodeParallely(s, beginIndex: 0, endIndex: n/4));
    });
    CompletableFuture thread2 = CompletableFuture.runAsync(() -> {
        result.addAll(encodeParallely(s, beginIndex: n/4, endIndex: n/2));
    });
    CompletableFuture thread3 = CompletableFuture.runAsync(() -> {
        result.addAll(encodeParallely(s, beginIndex: n/2, (int) (n/1.33)));
    });
    CompletableFuture thread4 = CompletableFuture.runAsync(() -> {
        result.addAll(encodeParallely(s, (int) (n/1.33), n));
    });
    // process each chunk separately
    // combine result from each thread
    while (true) {
        if (thread1.isDone() && thread2.isDone() && thread3.isDone() && thread4.isDone()) break;
    }
    return result;
}
```

RESULTS

SEQUENTIAL APPROACH	PARALLEL APPROACH
362 words 2,185 characters	362 words 2,185 characters
Total running time: 178 ms	Total running time: 45 ms
3KB -> 4KB, capacity increased :((3KB -> 2.8KB
Let's increase the input data size by 37 times!	
13,524 words 82,236 characters	13,524 words 82,236 characters
Total running time: 659 ms	Total running time: 164 ms
81KB -> 58KB => ~30% reduction in size	81KB -> 58KB => ~30% reduction in size

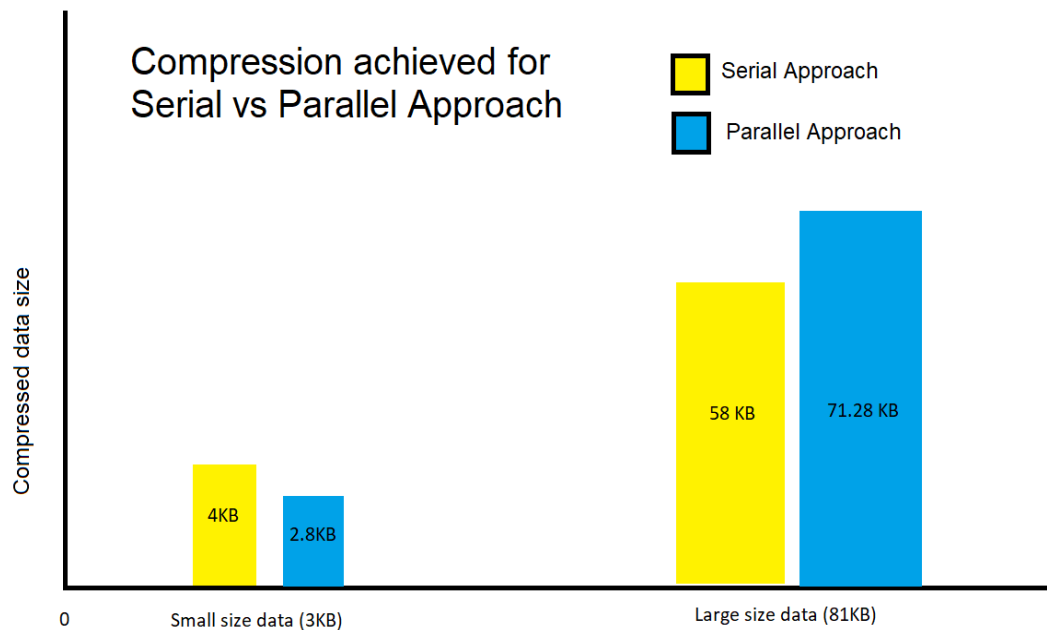


Fig: Compression Achieved

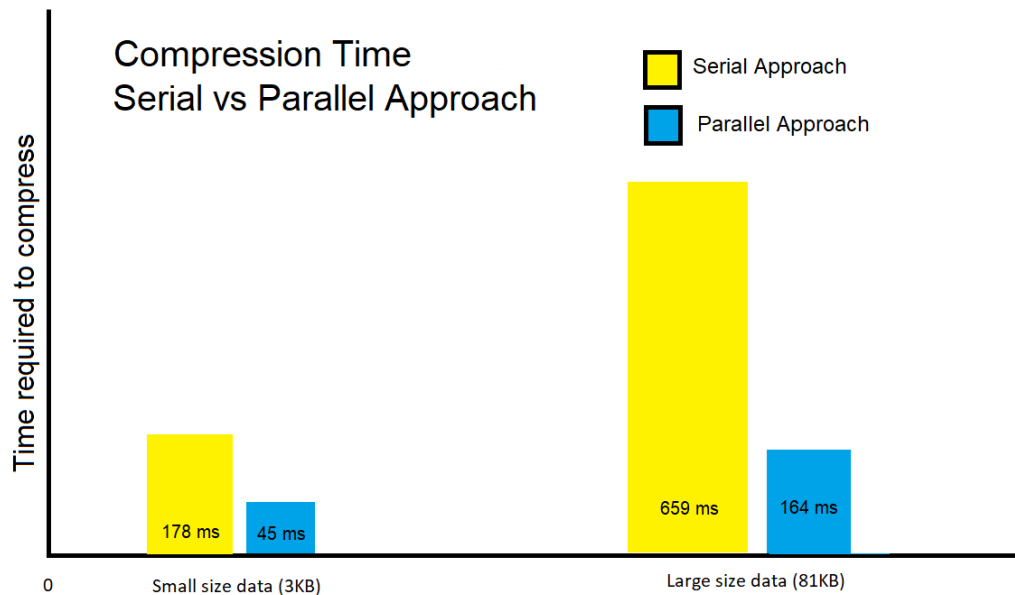


Fig: Compression Time

CONCLUSION

We evaluated two algorithms: the slower serial one, which uses a single processor and produces the compressed sizes and on the other hand is the parallel technique, which treats each block separately from the others and performs compression.

Due to the overhead of parallelization and limited dictionary size, the gain is obviously not anticipated to be 4-fold, but the time typically reduced by less than half. However, the compression gain in the serial model nearly completely disappears as the blocks get smaller, but with the new parallel processor structure, the compression performance declines much more gradually.

- Sequential approach is more time consuming but gives the best compression.
- Parallel approach provided a boost in speed but it came at a cost of compression quality.
- For parallel, 75% Reduction in time to compress, But only 12% compression is achieved.

We come to the conclusion that the serial structure though slow allows us to significantly reduce size of the strings without incurring too much of a loss in compression quality whereas a large number of processors allows for a faster compression but a lower compression ratio.

REFERENCES

1. Klein, Shmuel Tomi, and Yair Wiseman. "Parallel lempel ziv coding." *Discrete Applied Mathematics* 146.2 (2005): 180-191.
<https://u.cs.biu.ac.il/~wisemay/dam2005.pdf>
2. D. Belinskaya, S. DeAgostino and J. A. Storer, "Near optimal compression with respect to a static dictionary on a practical massively parallel architecture," *Proceedings DCC '95 Data Compression Conference*, 1995, pp. 172-181, doi: 10.1109/DCC.1995.515507.
<https://ieeexplore.ieee.org/document/515507>
3. Y. Zhang, X. Li, D. Hua, H. Chen and H. Jin, "An Lidar data compression method based on improved LZW and Huffman algorithm," *2010 International Conference on Electronics and Information Engineering*, 2010, pp. V2-250-V2-254, doi:10.1109/ICEIE.2010.5559775.
<https://ieeexplore.ieee.org/document/5559775>
4. M. Safieh and J. Freudenberger, "Address space partitioning for the parallel dictionary LZW data compression algorithm," *2019 16th Canadian Workshop on Information Theory (CWIT)*, 2019, pp. 1-6, doi: 10.1109/CWIT.2019.8929928.
<https://ieeexplore.ieee.org/document/8929928>
5. R. N. Horspool, "Improving LZW (data compression algorithm)," [1991] *Proceedings. Data Compression Conference*, 1991, pp. 332-341, doi: 10.1109/DCC.1991.213347.
<https://ieeexplore.ieee.org/document/213347>