EE 451 Final Project
Implementation and Analysis of Parallel Lempel-Ziv-Welch Algorithm

Fook Yen Edwin Chan
Yizhou Sheng
Mengwei Yang

Table of Contents

Introduction

Data compression is a principal technique in system design and communication. It aims at reducing the size of data without losing or only losing a small portion of the information stored in the data and allows (at least most) of the original data to be retrieved through a corresponding decompression. Many compression algorithms come into existence since the beginning of modern computer systems. One of them is the Lempel–Ziv–Welch (LZW) algorithm, developed by Abraham Lempel and Jacob Ziv and improved by Terry Welch. The algorithm is easy to understand and takes a relatively small effort to implement, while capable of achieving high performance: a well-formatted English book can be reduced to half of its original size with a single pass of the algorithm. Parallelizing the algorithm, however, is hard due to its input-adaptive nature.

We will investigate two different approaches to parallelizing the LZW algorithm: the straight-forward source division approach, and a more complicated approach which makes use of initial data layout and correspondence between the processors. We will compare their performance against the serial baseline, analyze the algorithms based on their different designs, point out the advantages and disadvantages of the algorithms, and provide potential modifications to improve the two algorithms.

Context

I. Background on Compression

The LZW algorithm is a variation over the earlier LZ77 and LZ78 compression algorithms. The LZ family of compression algorithms are adaptive, lossless algorithms for data compression. By lossless, it means that no information will be lost during compression. By adaptive, it means that the LZ algorithms would encode the input file in a single pass. It is constantly changing its encoding formula to generate the most efficient encodings for patterns that repeatedly appears in the input. There is thus no uniform encoding lookup tables for these algorithms, and the decoding method should also be adaptive. This is in contrast to static compression methods, which either have a universal way for decoding and encoding or require an initial pass of the input to determine certain characteristics of the input.

The LZ algorithms also hold a preferred property: the full encoding formula of the input file doesn't need to be stored or communicated to the decoder. The decoder can figure out the encoding rules used in the compressed data by sequentially decompressing the data, and the encoding rules can be thus be regenerated. The only information the encoder and the decoder need to agree on is an initial set of encoding rules, which will be illustrated in detail.

LZW was favored in the 1980s for its simplicity and high efficiency and soon became the most popular compression algorithm for file archives and images. It was later replaced by DEFLATE, which combines LZSS, another LZ variation, with Huffman Coding and many other layers of compression. LZW has ever since been restricted to images such as GIF and TIFF and an alternative compression method for the Adobe PDF format.

II. The LZW Algorithm

The basic logic behind the LZ algorithms is: if this is the first time the program encounters a certain pattern, it assigns a code to represent the pattern. The next time the program encounters the same pattern, it uses the code from before to replace the same pattern. The LZW algorithm formalized the above idea by storing the codes into a dictionary. The high-level description of the algorithm would be:

1. **Initialize the dictionary and assign a unique code to all characters of length one. This can either be done through reading the file once or if we know the encoding of the file beforehand (e.g. ASCII) we can assign a code to each possible character.**
2. **Read the file sequentially. Find the longest pattern W (substring) that matches an entry in the dictionary.**
3. **Output the corresponding code of W.**
4. **Add the character e immediately following W to W, and assign a new code for this new pattern W'.**
5. **Continue reading from the character following the substring (e). Go to step 2.**

   A more detailed explanation of the algorithm can be found in Dheemanth's paper in references.

The runtime of the algorithm is O(n), where n is the size of the file (since we only need to read the file once or twice). At first, it seems that the space complexity is also O(n) since the dictionary would store all unique characters in an encoding scheme (which is a constant) and the total length of the patterns is less than twice the size of the input file. However, the exact number of codes to represent the patterns, as well as the final compression ratio, depends on the input. In the worst case, there is never a repeated pattern, even of length 2. Then the dictionary would store the n different patterns of length 2, and the number of codes required to store each code would be log(n). If log(n) goes larger, the encoded file size would go large as well (up to n*log(n) bits). On the other hand, if the same pattern appears from time to time in the input file, and the pattern is discovered by the algorithm, then the number of codes required for compression would be minimal.

Throughout the rest of the report, we will define the **compression ratio** as **(the size of the original file / the size of the compressed file)**. The larger the compression ratio, the more efficient the compression algorithm is.

The corresponding decompression algorithm runs in a similar fashion:

1. Construct the same initial dictionary as the encoder, except this time the key is the code and the value is the character pattern being encoded (an inversion of key and value). This is the only information the encoder and the decoder have to agree upon.
2. Read in the first code P and output its corresponding string S(P);
3. Read in the next code C; Decoder knows that at this time, the encoder is able to recognize P, but cannot recognize P plus the next character w, and thus it would assign the next code to P+w; (break if there is no more code)
4. Decode C. If at this time we cannot recognize C, then C must come from the string immediately before it (P), then S(C) = S(P) + S(P)[0]. Otherwise, we get S(C) from the dictionary;
5. Emit S(C), assign the next available code to S(P) + S(C)[0] and add it to the dictionary.
6. Go to step 3;

Same as the compression algorithm, the decompression process requires only one pass of the input and completes in O(n) time, where n is the number of codes in the compressed file. Unlike the compression

algorithm, it doesn't depend on the pattern or layout of the compressed code (unless the underlying data structure relies on runtime information).

III. Challenges with Parallelization

The LZW algorithm itself is simple. To parallelize the algorithm, however, is not a straightforward task. There are several characteristics of the LZW algorithm that makes parallelization difficult:

a. The algorithm is Simple
   The serial algorithm requires a single pass of the input file and already runs in O(n) time. There are no recursive functions or deep loops that are "embarrassingly parallel". This means that besides the main while loop, there's not too much space for parallelization. Any attempt to optimize the process that runs over O(n) would result in a rise in complexity, thus become infeasible.
b. The Algorithm is Adaptive
   In other words, there are strong dependencies between every two steps in this algorithm. If we read the input out of order, we may overlook the effect the early characters and patterns have on the later inputs and produce different output than the sequential result. This can have a negative impact on the correctness and/or the efficiency of the program.
c. Specific Decoders is Required
   Due to challenge b), there is a high chance we will get a slightly different result from our parallelized LZW algorithms, either including some control signals or requires special reading order. In order to verify the correctness of our algorithms, we will need to create a specific decoder for each parallelized LZW encoder that produces different results. At the same time, a parallelized decoder would also provide better overall performance for the whole compression-decompression process.

There are several technical difficulties for implementing the full LZW algorithm, including bit-stream manipulation, file encoding conversion, and memory management. To emphasize on parallelization, we will avoid these complications and instead work on the core LZW algorithm. We will discuss the programming model, metrics and simplification in the next few paragraphs.

Objective

1. Implement a serial version of the LZW compression and decompression algorithm in C++. The serial program serves as a baseline of the comparisons and a reference of correctness.
2. Two different parallelization methods from past research are studied. They are then implemented in C++ in similar code structures and the same model. Originally, we chose the message-passing model, but after careful evaluation of the two papers, we decide to use Shared Memory models to simplify the implementation and ensure consistency between programs.
3. Analyze the two parallelization methods and give a hypothesis over the results based on the following metrics:
    a. **Execution time (Speed);**
    b. **Compression ratio (Efficiency or Quality);**
    c. **Best case and worst-case inputs for each algorithm.**
4. Collect or generate data sets and run them with the three programs. Compare the performance of the three algorithms by their execution time, compression ratio, and experiment over different data types and configurations (number of processors/data divisions).

To ensure consistency over the three programs and to keep the focus on the impact of parallelization, the following modification is made over the original LZW algorithm:

1. The full LZW algorithm would store the encoded data in binary bits. E.g. code 254 would be stored in 8 bits (11111110). To verify the result and avoid bitwise manipulation, we would instead store one code each line in an ASCII string format (e.g. "254\n") and use the number of bits assigned to each code and the total number of output code to calculate the best-case output size. We also assume that every output code will be stored with the same number of bytes. To calculate the estimated size, we use:

    $$\log_2(\text{the largest output code produced}) \times (\#\ \text{of codes written})$$

    to calculate the best-case output size.
2. We would restrict our file input to ASCII files, which has a reasonable total number of characters and easily parsable with C++. We will collect different types of ASCII inputs to simulate the process of compressing different data types. The performance of LZW focuses more on the pattern and layout of input data, rather than the file format. We would thus create an initial dictionary of size 256 for each algorithm.
3. We only record the time it takes to run the compression algorithm. Other system calls, such as File IO, is excluded from the time computation. Time used for thread controlling, such as thread creation and locking, however, are included since they are overheads due to parallelization.

Parallelization Hypothesis and Key Idea

1. Data Division (Standard) Parallelization

We found that the limit of the LZW compression technique is that each block is dependent on the previous block. Breaking down the input data into several blocks would lose the dependency between different data blocks. However, the algorithm would still be correct if we keep a separate dictionary for each data block, and during decoding, we simply decode each block separately from their starting point. In this standard parallelization technique, the divided strings are assigned to different threads to do the encoding process and the decoding process, as shown in Algorithm 1 and Algorithm 2.

Algorithm 1 The LZW Standard Parallel Encoding Algorithm

Input: S, Data File
Input: N, Number of divided blocks
Block size $= \frac{length\ of\ S}{N}$, denoted as B
Divide index = thread ID $*$ block size
for $n = 1$ to N **par do**
  Initialize the encode dictionary for each thread
  Assign data S_n and Block size B_n to thread n
  // Encoding process
  // P represents previous string and C represents current string
  for $i = 1$ to B_n **do**
    Read new string from S_n and add it into C
    if $P + C$ is in dictionary **then**
      $P = P + C$
    else
      Encode P and then append it into output encoded string E_n
      Set a symbol mapping for $P + C$ in the dictionary
      Update $P = C$
    end if
  end for
  Add -1 and the size of E_n in the header of each output string E_n
end for
Output: $E = \{E_1, E_2, E_3, \ldots E_n\}$

In algorithm 1, we can see that the data is split into different blocks, and users define the number of blocks. Compared with the serial LZW algorithm, this parallel technique completely discards the dependency between data blocks. Each thread can encode their string block independently, and each thread has its own encode dictionary. Also, each thread knows the start index, end index, and block size of the input data file. A constant-size block marker is added into the head of each block to identify the starting position for each decoding thread, or we can store each encoded block in separate files.

Algorithm 2 The LZW Standard Parallel Decoding Algorithm

Input: $E = \{E_1, E_2, E_3, \ldots E_n\}$

Derive number of threads N and length E_n^l of each E_n from input
for $n = 1$ to N **par do**
  Initialize the decode dictionary for each thread
  Assign encoded string E_n and length E_n^l to thread n
  // Decoding process
  // P represents previous string and C represents current string
  Read the first string from E_n, decode it and then output into decoded string D_n
  for $i = 1$ to $E_n^l - 1$ **do**
    $P = C$
    Read new string from E_n and add it into C
    if C is in dictionary **then**
      Decode C and then append it into output decoded string D_n
      P' = P, C' = the first word of C
      Set a symbol mapping for $P' + C'$ in the dictionary
    else
      P' = P, C' = the first word of C
      Set a symbol mapping for $P' + C'$ in the dictionary
      Decode $P' + C'$ and then append it into output decoded string D_n
    end if
  end for
end for

Output: $D = \{D_1, D_2, D_3, \ldots D_n\}$

In algorithm 2, it describes the process of decoding. Each thread finds its string block through the block marker. Also, all threads will figure out their respective decode dictionary. Thus, the encoded strings in algorithm 1 can be decoded concurrently.

We would refer to this algorithm as the "standard parallelization" algorithm, as the data division idea is applicable to most compression algorithms, and the de-facto process used for most compression programs.

Advantages of this method:

1. The algorithm makes full use of parallelization. There is no data dependency between threads, thus it requires no locks or conditions. All processors can start at once, and only need to synchronize before the final output. This means a PRAM model is applicable to modeling the algorithm, though our implementation sticks to PThread to keep consistent with the second algorithm.
2. The algorithm is easy to implement. We only need to specify the beginning and the end of each data block, then each thread can handle its respective block using the serial code. Same for decoding.

Disadvantages of this method:

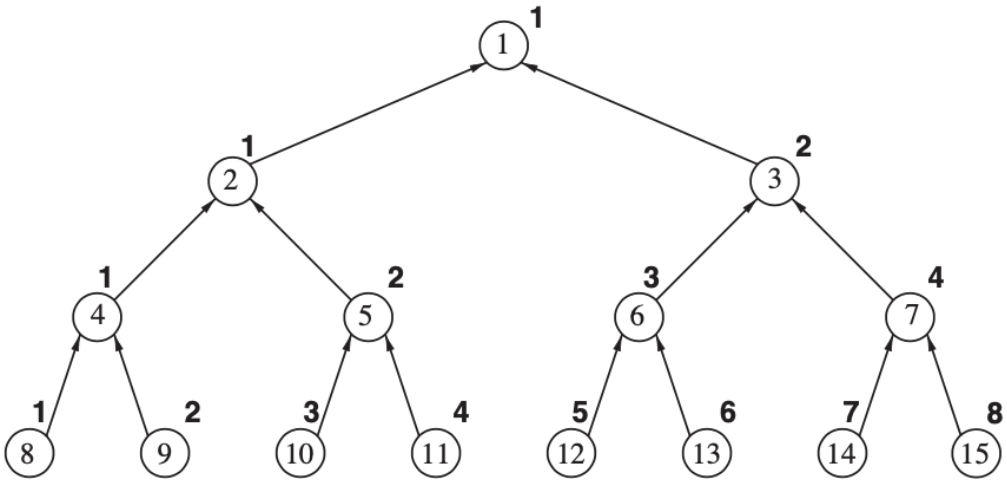
1. Dropping dependencies means that the later code blocks will miss some of the patterns already discovered in previous blocks, leading to repeated encoding into its dictionary. There will be an increase in the total number of codes in the output, which negatively impacts the compression ratio.
2. If the data blocks happen to break a long pattern that can be encoded into a single code, it will have a large impact on the compression ratio.

These issues will become significant when the file is small, where the algorithm is more likely to break a pattern or cause repeated entry into the dictionary. For a larger file or data with few or no long patterns, these issues would be minimal.

The speedup should be close to **P**, which is the number of threads used given enough processors.

2. Partially-Dependent Data Division

To avoid repetitive entries or pattern breakdown due to loss of dependencies, a newer algorithm proposed in 2005 introduces a "partially dependent" data layout in parallelizing LZW. This algorithm starts by dividing the input data into different partitions, but instead of running separate encoding threads at the same time, the algorithm would first align the blocks in a full binary tree, illustrated below:



The number inside each node denotes the id of the data block, with #1 being the first block of data derived from the original input. The data is then processed in a top-down order of the tree: data block #1 is first encoded with the standard LZW process. After #1 finishes, it sends independent copies of its dictionary to node #2 and #3. The two nodes then start processing their data using the dictionary from #1. It will look up existing patterns and insert new patterns into their own dictionary. After completing each of them passes on their dictionary to two more nodes, which initialize their independent processing. From observation, node #i would receive the dictionary from node **#i/2** (integer division) and sends its dictionary to node **#i*2** and **#i*2+1** after completion, if they exist.

To avoid excessive copying of a dictionary, we observe that nodes on the same branching direction can be processed using one single thread/processor. This conclusion can also be drawn by understanding the above diagram as a task dependency graph. The maximum level of parallelization is equal to the number of leaves, or approximately half the number of blocks. Below is one way to assign each data block (task) to the processors. The same dictionary can be reused by the same processor multiple times when processing the left child, but a copy needs to be made for the right child, which is handled by a new thread.

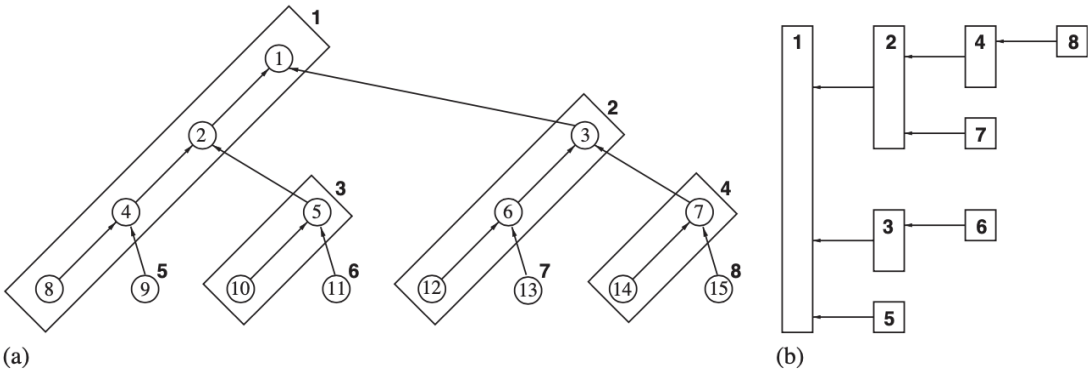


Fig. 3. New hierarchical structure: (a) Tree of blocks; (b) Tree of processors.

The processor that is responsible to process node **#i** can be derived from the following formula:

$$f(i) = \frac{1}{2}\left(\frac{i}{2^{r(i)}} + 1\right)$$

The pseudo-code proposed for the algorithm in the paper is as follows:

The expected runtime of this algorithm should be O(n/b * log(b)), where b is the total number of data blocks. This is drawn from the fact that all tasks at the same level can be performed at the same time. We predict the compression ratio should be lower than the serial version, but higher than the standard parallel version given the same number of data blocks. We also predict that this algorithm, given the same number of processors, would require a longer time than the standard parallel version.

We decided to use the PThread programming model. In a real implementation, certain changes are made to accompany the modern C++ code structure and compiler features:

1. The original algorithm suggests inserting all codes into the same, shared dictionary and differentiates entries from different blocks by adding a “timestamp” (or block id). This approach would avoid copying the dictionary, but in exchange asks for a lock-controlled dictionary shared by all threads. Also, considering that the C++ <unoredered_map> is implemented with a “fixed-bucket” approach, the access time of the dictionary would be linear as the dictionary grows. We stick to copying the dictionary since multiple small dictionaries are better than one clunky dictionary in the long run.
2. To make sure the same processor will go on to process the left child of the current block, instead of invoking two new threads at the end of the algorithm, we instead initialize one

thread for the right child, then recursively call the current function with the parameters of the left child. The dictionary of the current thread is reused for the recursive call (by using a shallow copy of the dictionary pointer. See code for details).

Advantages of this algorithm:

1. A reasonable level of dependency is kept between data blocks, contributing to less repeated entries and a higher compression ratio.
2. The layout is highly customizable. We can use higher-level trees to increase the level of parallelization and thus achieve higher speedup. In exchange, we will ignore some dependencies between blocks and downgrades a little on the compression ratio.

Disadvantages of this algorithm:

1. The speedup achieved is lower than the first algorithm, given the same number of processors. This is due to task dependencies and a finer grinding in the data division.
2. The workload is imbalanced, meaning that the algorithm is not suitable for all parallel models. An implementation in PRAM, for example, may turn out to have many idle processors in an iteration and thus not exploiting the full power of the machine.

Experimental Setup

We have selected a variety of test cases, including ASCII art, long English texts, specialized cases, edge cases and run it with the versions that we have created, time the results and verify the results by taking the bit differences against the original file. The correctness of the encryption will also be verified against the serial version.

For test cases, we have included the following:

ASCII Art:

Mario (Small)
Eiffel Tower (Medium)
Generated ASCII art (Large)

Specialized Cases:

Strong Pattern Text
Weak Pattern Text
Random Text

Long English Texts:

English Bible Text
Sherlock Holmes Adventure
Pride and Prejudice

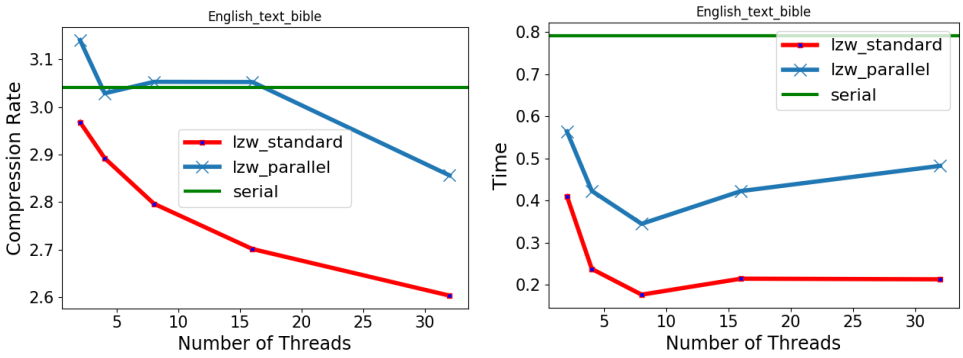
By providing a variety of inputs, we can have different benchmarks, which are compression rate and the time required for compression and extraction to indicate the performance and then we can show how the difference in inputs of data will affect the compression rate of the data and the timing of it.

All executions are performed on a desktop computer with an 8-core Intel i7-9700K processor with 16GB RAM.

Results and Analysis

I. Sample Input: English Bible

We first run our program on the English Bible (King James version), which is the same test case used in the 2005 paper. We run all three programs with different configurations and record the compression rate and the time.

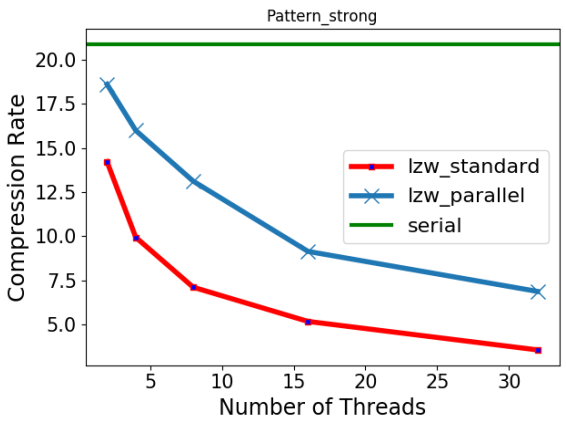


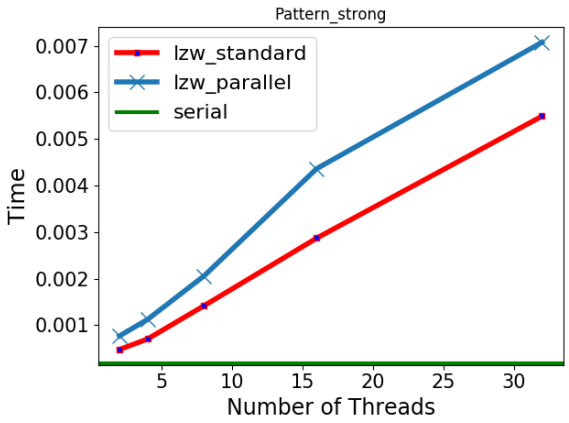
The result basically agrees with our assumption: The parallel algorithms run faster than serial, with the standard parallel algorithm outperforming the partially-dependent algorithm in speed. The partially-dependent algorithm has a higher compression rate than standard parallelization. The compression rates of both parallel algorithms decade as the number of data blocks grow, since more dependencies are lost, and more repeated entries are made. The serial program generally has the best compression rate, however in this test the partially-dependent algorithm sometimes have a higher compression rate. We will explain this phenomenon in a later example.

II. Specialized Input

A. Strong Pattern

LZW algorithm is at its best if the input has a strong pattern. The strongest pattern can think of is the repetition of a single character. In this example, we arbitrarily create an input consisting of 1024 ‘a’s and send in for compression.



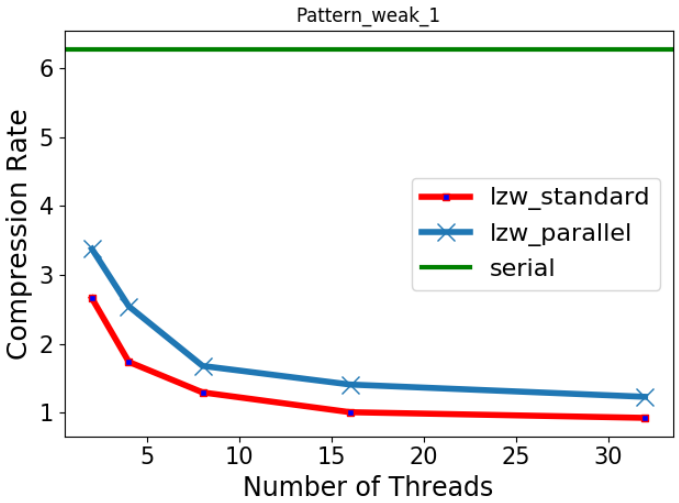


The serial program can observe very long patterns and thus has an amazing compression rate. The parallelized algorithms, due to loss of dependency and repetitive entries of shorter patterns, have worse compression rates as the number of data blocks grow. It is also notable that the speed performance of parallel algorithms goes down as the number of threads increase, mainly because of the thread creation overhead dominating the time complexity. We will see that with larger input, the thread overhead would have less impact on runtime.

B. Weak Pattern

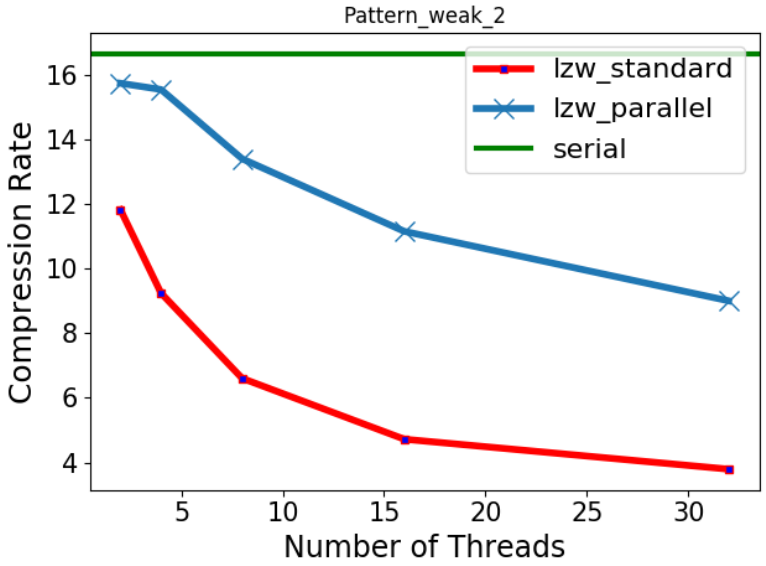
We predict that if the border of data blocks lies in a long pattern, there can be a significant impact on the compression rate. Also, the partially-dependent algorithm should have a better chance of preserving the long pattern than the standard algorithm. To verify, we created two weak patterns:

Pattern 1: an increasing iteration of the 26 lower-case English characters, in the form of **"aababcabcdabcde…"** with each iteration including once more character than before. The pattern can be picked up only by the serial program.



The experiment outcome agrees with our hypothesis, and both parallel algorithms miss the pattern largely. Still, the partially-dependent algorithm performs slightly better than the standard algorithm.

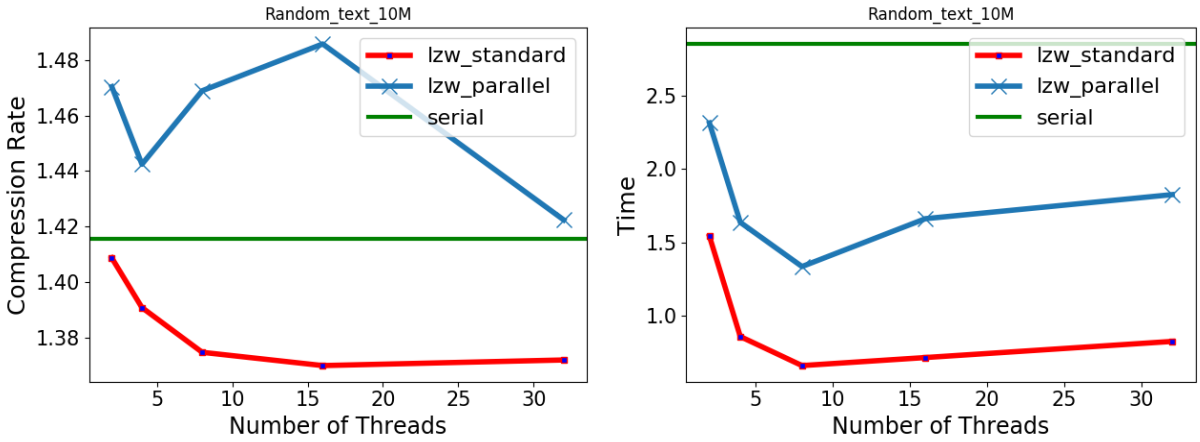
Pattern 2: a complete iteration of the 26 lower-case English characters, repeated for 1024 times. The pattern is long and can be discovered by independent threads if it is not cut in the middle.



The results show that the partially-dependent algorithm performs significantly better than the standard parallelization in figuring out the pattern and thus have higher efficiency.

C. Random Text

A random text of size 10M is generated with the std::mt19937 uniformly distributed generator. This is the worst-case input for LZW, as there is hardly any pattern in the text.



The compression rate of all three algorithms is around 1.4 ~ 1.5, which is by no means plausible. The data does show that parallel algorithms will show a

higher speed up in larger files since the thread overhead is small compared to file size.

III. English Text

English texts have a reasonable number of patterns, made up of repeated words or phrases, and this is a suitable data type for LZW compression. Apart from the English Bible, we also run the algorithms on two English books, *Pride and Prejudice*, and *The Adventures of Sherlock Holmes.*

IV. Image (ASCII Art)

We use ASCII art as our image input. ASCII art simulates the data layout of real images: pixels of the same color are usually located next to each other, and many pixels share the same color (the background). We run our input on three ASCII text files, each with a different style and size.

Full results can be found in the appendix of the report. Complete statistics can be found in the Excel sheet provided in the project archive, as well as the original input data and execution logs.

Conclusion

With the results from the experiment, we can conclude the following facts of parallelized LZW algorithms:

1. In terms of speed on large input data, parallel LZW algorithms are better than serial LZW. The Standard Parallelization method is faster than the Partially-Dependent method, given the same number of concurrent threads.
2. The serial LZW has the best compression rate. The Partially-Dependent method effectively preserves data dependency and has better compression than the Standard method.
3. For both parallel algorithms, the more threads/data blocks used, the faster the parallel algorithms will become. The speedup grows close to linearly (given the thread numbers do not exceed the number of logical processors), with the exact growth rate depending on the input. In exchange, the compression rate decreases almost logarithmically, with exact values depending on the input layout.
4. Both parallel algorithms perform better on organized inputs, similar to the serial program.

Explanations for irregular cases:

1. When the input data has a small size, the thread controlling overhead can dominate the time consumption, which results in a downgrade in runtime. The more thread being created, the longer it takes for the parallel programs. Parallel programs would start performing better when the input size gets larger (>100KB).
2. The Partially-Dependent method sometimes achieves a better compression rate than the serial version, a discovery not present in the 2005 paper. This is because the total number of assigned codes is smaller for the Partially-Dependent method. Take the complete results for the English Bible as an example:

```
sanithovski@DESKTOP-PV9E79H:/mnt/f/ee451/EE451FinalProject$ ./lzw_serial_encode da
ta/english_text_bible.txt  SerialBible.out
Encoding
men.    20165
Largest code assigned: 609852
Number of bits to store each code: 20
Number of output codes: 609596
Estimated best-case compressed size: 1523990 bytes
Encoding time = 0.773117 sec
Input file size = 4634229
Estimated Compression Rate = 3.04085
sanithovski@DESKTOP-PV9E79H:/mnt/f/ee451/EE451FinalProject$ ./lzw_parallel_encode
3 data/english_text_bible.txt ParaBible.out
Encoding time = 0.567568 sec
Largest code assigned: 425137
Number of bits to store each code: 19
Number of output codes: 621284
Estimated best-case compressed size: 1475549 bytes
Estimated Compression Rate = 3.14068
```

We can see that although the Parallel algorithm does generate more output codes, it has a smaller dictionary and requires a smaller number of bits to store each of the output codes. The total output

size is thus reduced. This is a coincidence when the total number of encoded patterns lie on the boundary of two different powers of 2, and mainly occur with the Partially-Dependent method.

With the above conclusion, we have found a way to control the trade-off between speed and compression rate, by manipulating the parameters of the two parallel algorithms. We can also derive a preferred configuration for different inputs and outputs:

1. If the input file is small, stick to the Serial LZW.
2. If the input file is large and we want maximum speed for compression, use the Standard Parallelized LZW.
3. If we want to achieve a relatively fast speed while also maintaining a high compression rate, we can use the Partially-Dependent Parallelized LZW.

References

Welch, "A Technique for High-Performance Data Compression," in *Computer*, vol. 17, no. 6, pp. 8-19, June 1984.
doi: 10.1109/MC.1984.1659158
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1659158&isnumber=34743

Dheemanth, H.N. "LZW Data Compression" in American Journal of Engineering Research (AJER) e-ISSN : 2320-0847 p-ISSN : 2320-0936 Volume-03, Issue-02, pp-22-26
URL:http://www.ajer.org/papers/v3(2)/C0322226.pdf

"LZW (Lempel–Ziv–Welch) Compression technique," GeeksforGeeks, 09-Sep-2019. [Online]. Available: https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/. [Accessed: 05-Dec-2019].

Mishra M K, Mishra T K, Pani A K. Parallel Lempel-Ziv-Welch (PLZW) technique for data compression[J]. Int. J. Comput. Sci. Inf. Technol, 2012, 3(3): 4038-4040.
URL: https://pdfs.semanticscholar.org/be7c/b6d0d3935337fd633dc9b3667bf28904c891.pdf

Appendix: Complete Experiment Statistics

List 1: Estimated Compression Rate of all three algorithms

Estimate Compression Rate	Serial	Standard, 2 threads	Standard, 4 threads	Standard, 8 threads	Standard, 16 threads	Standard, 32 threads	Parallel, 3 blocks	Parallel, 7 blocks	Parallel, 15 blocks	Parallel, 31 blocks	Parallel, 63 blocks
ascii_art_large	10.9839	10.1065	10.0705	9.11061	8.70468	8.19787	10.7754	10.1271	9.99861	9.77193	8.65089
ascii_art_med	6.02961	5.41003	5.2317	4.53297	4.35729	3.69877	5.76549	5.95969	5.40461	4.8183	4.61831
ascii_art_small	6.36737	5.39607	4.52431	4.18906	3.3934	2.71499	6.13489	5.7228	5.12627	5.05347	4.21533
english_text_bible	3.04085	2.96828	2.89254	2.79674	2.70147	2.6035	3.14068	3.02905	3.05318	3.05278	2.85645
english_text_lorem_ipsum	1.66341	1.57415	1.37085	1.34716	1.19283	1.05901	1.60471	1.65133	1.5369	1.42306	1.42603
english_text_pride_and_prejudice	2.58249	2.46496	2.35029	2.23991	2.1396	2.04555	2.67526	2.57044	2.59058	2.58909	2.39803
english_text_sherlockholmes	2.61413	2.45512	2.39401	2.31635	2.23647	2.15984	2.56002	2.59844	2.47596	2.47762	2.3283
pattern_strong	20.898	14.2222	9.94175	7.11111	5.17172	3.55556	18.6182	16	13.1282	9.14286	6.87248
pattern_weak_1	6.26786	2.65909	1.73762	1.2952	1.00862	0.926121	3.375	2.54348	1.67943	1.40964	1.23158
pattern_weak_2	16.6504	11.8224	9.24444	6.60645	4.72057	3.79259	15.7445	15.5514	13.4058	11.1584	9.00677
random_text_1024	1.12527	0.981783	0.997079	0.947271	0.920036	0.905393	1.07789	1.01688	1.07001	1.02196	0.975238
random_text_10M	1.4155	1.40865	1.39079	1.37467	1.36989	1.37191	1.47035	1.4424	1.46889	1.48575	1.42236
random_text_1M	1.34862	1.34597	1.34536	1.33214	1.30738	1.28621	1.40408	1.37736	1.427	1.47267	1.40911
random_text_4096	1.17263	1.14318	1.11577	0.997808	1.01436	0.959251	1.25031	1.19661	1.23114	1.12993	1.04144

List 2: Core algorithm runtime of all three programs

Runtime	Serial	Standard, 2 threads	Standard, 4 threads	Standard, 8 threads	Standard, 16 threads	Standard, 32 threads	Parallel, 3 blocks	Parallel, 7 blocks	Parallel, 15 blocks	Parallel, 31 blocks	Parallel, 63 blocks
ascii_art_large	0.0287206 sec	0.0147987 sec	0.0081518 sec	0.0081153 sec	0.0075393 sec	0.0090552 sec	0.0201947 sec	0.0157523 sec	0.0137265 sec	0.0166212 sec	0.0196376 sec
ascii_art_med	0.0010796 sec	0.0008854 sec	0.0009611 sec	0.0020182 sec	0.0031346 sec	0.0060503 sec	0.0013322 sec	0.0015756 sec	0.0029951 sec	0.0050347 sec	0.0095962 sec
ascii_art_small	0.0005345 sec	0.0006657 sec	0.0009901 sec	0.001782 sec	0.0033432 sec	0.0054045 sec	0.0010282 sec	0.001345 sec	0.0023126 sec	0.0049241 sec	0.0072156 sec
english_text_bible	0.791363 sec	0.41039 sec	0.237769 sec	0.176792 sec	0.214726 sec	0.213225 sec	0.564097 sec	0.423048 sec	0.344918 sec	0.422758 sec	0.482596 sec
english_text_lorem_ipsum	0.00043 sec	0.000698 sec	0.000831 sec	0.001647 sec	0.002993 sec	0.005237 sec	0.000924 sec	0.001289 sec	0.002221 sec	0.004570 sec	0.007582 sec
english_text_pride_and_prejudice	0.0898484 sec	0.0554972 sec	0.0350802 sec	0.0351335 sec	0.0430143 sec	0.0458255 sec	0.0730558 sec	0.0644415 sec	0.0684733 sec	0.0936853 sec	0.106882 sec
english_text_sherlockholmes	1.34939 sec	0.714275 sec	0.393979 sec	0.286416 sec	0.307608 sec	0.330442 sec	0.919433 sec	0.763445 sec	0.581255 sec	0.635896 sec	0.822674 sec
pattern_strong	0.0001806 sec	0.0004759 sec	0.000704 sec	0.001417 sec	0.0028693 sec	0.0054887 sec	0.0007687 sec	0.0011244 sec	0.0020579 sec	0.004362 sec	0.0070765 sec
pattern_weak_1	0.0001254 sec	0.0004721 sec	0.0006351 sec	0.0015109 sec	0.00287 sec	0.0049516 sec	0.0007598 sec	0.0011114 sec	0.0023026 sec	0.0043254 sec	0.0070103 sec
pattern_weak_2	0.0022784 sec	0.0013697 sec	0.0011468 sec	0.0017326 sec	0.0032576 sec	0.0044639 sec	0.0021916 sec	0.0021964 sec	0.0030447 sec	0.0057668 sec	0.00872 sec
random_text_1024	0.0002572 sec	0.0005223 sec	0.000813 sec	0.0016372 sec	0.0028211 sec	0.0057533 sec	0.0008201 sec	0.0012693 sec	0.0024751 sec	0.0043093 sec	0.0079829 sec
random_text_10M	2.85017 sec	1.54126 sec	0.858971 sec	0.661435 sec	0.716141 sec	0.825974 sec	2.31218 sec	1.6361 sec	1.3354 sec	1.66107 sec	1.82491 sec
random_text_1M	0.185364 sec	0.112167 sec	0.0769778 sec	0.077122 sec	0.09015 sec	0.086361 sec	0.172908 sec	0.131255 sec	0.137275 sec	0.191284 sec	0.222311 sec
random_text_4096	0.0006023 sec	0.0007531 sec	0.0008522 sec	0.0018638 sec	0.003418 sec	0.0056676 sec	0.0011735 sec	0.0015528 sec	0.0028094 sec	0.0057633 sec	0.0081592 sec