Parallel Lossless Compressors

Huilin Xiong (hxiong)

Qier Li (qierl)

Recording Link: https://www.youtube.com/watch?v=1aNZsRPyAFw&ab_channel=QierLi

The project parallizes Huffman Coding and LZ77 compression algorithms with OpenMP, and conducts general analysis on the benchmarking and general industrial implementations.

Brief

Compression is one of the most widely-used applications in the industry, and is developed over decades. The most recent significant variant is Facebook's zstd, which was released on 31 Agust 2016, providing great compression ratios and reasonable compress/decompress throughputs. LZ4 is also heavily used for its highest throughput among popular variants.

Our motivation is to extremely parallelize the algorithm to improve the compress/decompress speed of one large file, which could lower the latency. Imagine the following case: transfer one large file through a fast-but-charge-by-traffic network (e.g., 5G network). Applying a parallel compression algorithm could greatly lower the latency of the process of compression and transfer, and save your pocket.

LZ77 and Huffman Coding are the 2 base algorithms repectively of Lempel-Ziv family and Entropy encoding with a long history. LZ77 is theoretically a dictionary coder, maintaining a sliding window during compression. Huffman Coding is a particular type of optimal prefix coder, representing data items with the frequencies of occurrences. Most modern compressors, such as zstd, are based on those 2 algorithms.

This project has learned and implemented this two most important lossless data-compression algorithm （sequential version), then designed and implemented parallel versions for each using OpenMP. The project further analyzed performance including compression ratio and throughput/speedups on PSC machines. By such a programming practice, the project developed some conclusions and assumptions for the parallism strategy and potentials for general compression algorithms and Facebook's zstd.

Resources

This project heavily refers to *Mastering Algorithms with C: Useful Techniques from Sorting to Encryption*, especially implementing the sequential versions of algorithms.

The algorithms is developed to be executable and configurable with timer in **C/C++, OpenMP**, and then test on mainly two types of files, that literature such as *War and Peace* and source codes such as Angular.js.

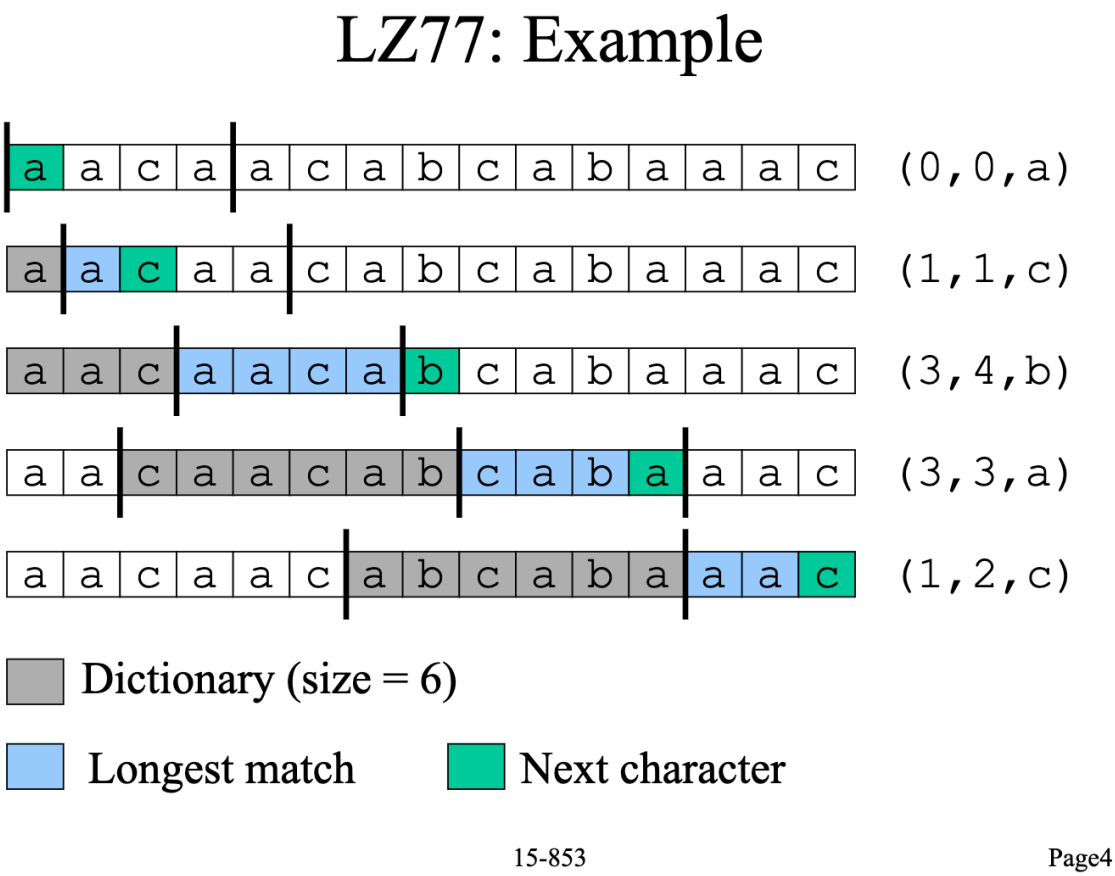
The tests is run on PSC machines sequentially and parallelly from num_threads 1 to 128.

LZ77

Intro

LZ77, short for Lempel-Ziv algorithm, is a lossless data-compression algorithm created by Lempel and Ziv in 1977. LZ77 has been adopted for many modern complicated algorithms such as Facebook's zstd.

Breakdown



the slide refered from 15-853 lesson

LZ77 compression process is a simple sliding-window process: from the start to the end, continually matching the longest sequence from buffer to the current window, and record the sequence in buffer as a (offset, length, ending character) tuple.

The sequential process contains the following steps (assuming the original file and encoded file could fit in memory):

1. Read the file into memory.
2. Initialize the sliding windows and starts encoding.
3. Write the encoded content to another file.

For most storage (disk, SSD...), reading/writing will not benefit from parallism (or even worse with parallism), so step 1 and step 3 are not parallelizable.

Generally the parallelizable of the process is the step 2. And as the file size grows, the ratio of time of step 2 increases. It could be concluded that large file is more parallelizable. And a linear-speedup of compression (throughput) is expected for large files.

The strategy of parallism is to **partition consecutive parts of original files, and then encode the partitions each as sequentially** and indicates the parallism in the headers. The pitfall for ratio/speedup is the following:

1. Partition might break a potential matched sequence and require more encoding tuples.
2. Extra padding is needed, as encoding is bit-level and partition and headers are byte-level.
3. Extra headers is needed, to indicates the number of partitions and positions.

Above 2 pitfalls just requires a few bytes to encode, which is negligible and the following tests prove it.

Deliverables

A executable program is developed and configurable based on the following strategy. The decoding process is not parallelized because it is comparably fast.

Inside the path lz77/src, there's a Makefile to compile a executable file LZ77.

```
% ./lz77 -h
./lz77: invalid option -- 'h'
Usage: ./lz77 -f <filename>
                [-x for extract]
                [-t <number of threads, 0 for sequential>]
```

Tag -t indicates the number of threads.

The tuple is configurable in the source file compress.h as following:

```
#define LZ77_TYPE_BITS   1 // indicates a matched sequence or a single character
#define LZ77_WINOFF_BITS 12 // [0, 4096]
#define LZ77_BUFLEN_BITS 5 // [0, 32]
#define LZ77_NEXT_BITS   8 // [0, 256]

#define LZ77_WINDOW_SIZE  4096 // must in [2, 2^LZ77_WINOFF_BITS]
#define LZ77_BUFFER_SIZE  32 // must in [2, 2^LZ77_BUFLEN_BITS]
```

As the window size and buffer size increases, the length of the matched sequence is likely to be longer, but it requires more bits to represent each tuple and also more time to compress. And it's highly trace-dependent. Based on the test files, the default configuration achieves a reasonable throughput and compression ratio.

Result

The records of benchmarking is shown below:

Correctness is checked by md5sum and diff command.

Compression ratio negligible for sequential and parallel results.

When the number of threads is below 16, the speedup is nearly linear to num_threads.

When the number of threads reach 128, the speedup stops increasing.

For files that has more duplicated content (angular.js), the compression ratio is much better.

war&peace(32M)				
Algorithm	Thread Num	Compress Time	Total Time (ms)	**Total Speedup**
lz77	1	21807.544	21844.794	1.000
	4	5467.133	5498.592	3.973
	8	2753.437	2792.868	7.822
	16	1558.221	1595.659	13.690
	32	962.479	1001.362	21.815
	64	422.977	453.289	48.192
	128	325.274	357.07	61.178
	Raw File Bytes	32022870	Compressed File Bytes	20156523
	Compression Ratio	1.59	Decode Time	851.717ms

angular.js (21M)				
Algorithm	Thread Num	Compress Time	Total Time (ms)	**Total Speedup**
lz77	1	9298.655	9319.683	1.000
	4	2458.891	2476.388	3.763
	8	1225.719	1252.558	7.441
	16	644.235	665.019	14.014
	32	338.801	361.162	25.805
	64	185.644	211.809	44.000
	128	201.124	220.842	42.201
	Raw File Bytes	22046527	Compressed File Bytes	9631119
	Compression Ratio	2.29	Decode Time	438.476 ms

Analysis

One of the reason that speedup can't increase linearly when number of threads increase is **uneven distribution**. Although we partitions the file into parts of same size, the **matching process** might requires different amount of scans for different positions. When the current sequence unmatches, the scan stops and continute to the next section. And it's nature for files having different amount of duplicated sequence at different positions.

By recording each time of OpenMP thread, we prove that uneven workload distribution is one of the reason that it fails to achieve linear speedup.

```
[qierli@r349 src]$ ./lz77 -f test/war_and_peace -t 32
Thread 31 compress time: 115.074 ms
Thread 6 compress time: 128.255 ms
Thread 19 compress time: 125.782 ms
Thread 20 compress time: 126.270 ms
Thread 15 compress time: 126.987 ms
Thread 3 compress time: 127.091 ms
Thread 29 compress time: 127.147 ms
Thread 21 compress time: 127.173 ms
Thread 18 compress time: 127.232 ms
Thread 4 compress time: 127.331 ms
Thread 25 compress time: 127.360 ms
Thread 23 compress time: 127.389 ms
Thread 27 compress time: 127.634 ms
Thread 16 compress time: 127.846 ms
Thread 28 compress time: 127.923 ms
Thread 2 compress time: 127.941 ms
Thread 7 compress time: 127.971 ms
Thread 24 compress time: 128.168 ms
Thread 1 compress time: 128.151 ms
Thread 13 compress time: 128.193 ms
Thread 9 compress time: 128.407 ms
Thread 17 compress time: 128.551 ms
Thread 8 compress time: 128.631 ms
Thread 10 compress time: 128.710 ms
Thread 26 compress time: 128.908 ms
Thread 22 compress time: 128.921 ms
Thread 5 compress time: 131.054 ms
Thread 12 compress time: 129.607 ms
Thread 30 compress time: 129.757 ms
Thread 14 compress time: 130.067 ms
Thread 11 compress time: 130.565 ms
Thread 0 compress time: 139.610 ms
Total time: 170.274 ms
Compress time: 147.862 ms
```

The reason that more duplicated content has better the compression ratio is the nature of LZ77 algorithms. With more duplicated content, tuples could reprsents longer sequence in the original files.

Huffman Coding Compression

Intro

Huffman Coding is a lossless data compression algorithm. It makes full of the frequency of each character in the file, and assign variable length bits to represent them, to reduce the total compression bit length. The most frequent character will have the shortest code, and the least frequent character will have the longest code. In this way, it transform the original file into a length-optimized file to reduce file size.

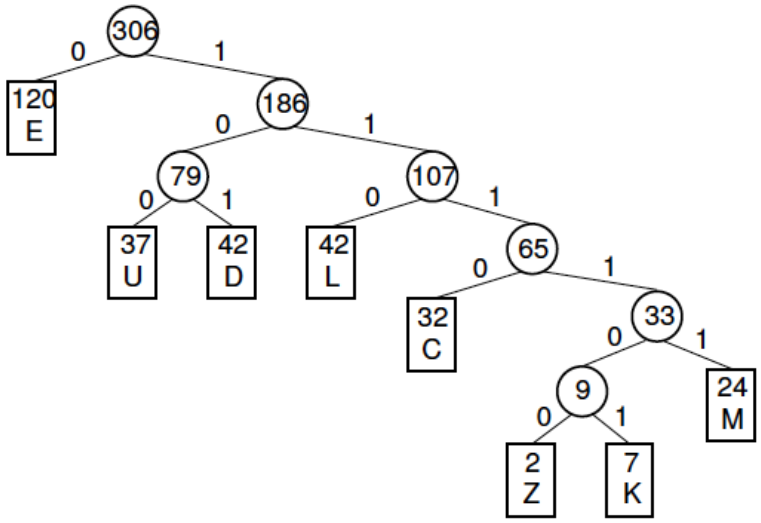
Breakdown

The basic steps of Huffman Coding is:

- Read file, and Count the character frequency
- Build the Huffman Tree based on character frequency
- Encode character with '0' and '1'
- Generate Compression for the file with encoded character
- Output compressed file

The most important part is building Huffman Tree, here is an example of the algorithm.

The example of Huffman Tree is like below. The leaf nodes are characters. After getting the frequency of each character, it will pick two nodes with smallest frequency (which is 'Z':2 and 'K':7 in this tree), and merge them together (Node 9=2+7). Then keep this process until there is only one root node, which is 306 in the figure. Then assigning the code as node→left = '0', node→right='1', the final code of each The code of each character is the path from the root to its leaf node. For example, the code of 'E' is '0', and the code of 'C' is '11100. So if we want to compress a sentence like "EEEEC"(5 bytes = 40 bits), we just use "000011100" (9 bits).



(refer to https://cgi.luddy.indiana.edu/~yye/c343-2019/huffman.php)

Parallel Design

After finishing the sequential Huffman Encoding, we test the compute time of each part, the result is as below:

```
[xiongh@bridges2-login011 HuffmanCoding]$ ./main e w1.txt w1_output.bin w1_map.txt 1
read_raw_file, size 3202286
*** read_raw_file time= 0.00641633
*** build_huffman_tree time= 2.4476e-05
*** encode time= 4.8811e-05
Target output bitsize = 14390112
*** output time= 0.121394
***** Total Time= 0.12854
OK
```

We can see that output and read_file are the top2 longest part of the whole process. Other process like building the tree is trivial to performance. When we breakdown the output() function into generate_result() and output_to_file(), we found generate_result() is the longest part.

```
[xiongh@bridges2-login011 HuffmanCoding]$ ./main e w1.txt
read_raw_file, size 3202286
*** read_raw_file time= 0.00641324
Target output bitsize = 14390112
*** generate encoded file time= 0.172529
*** output time= 0.00847037
***** Total Time= 0.18754
```

So we identify the bottleneck of Huffman Coding as generate_encoded_file(), and read_raw_file(), and we will use faster it by parallel execution.

Parallel Strategy: Partition the file

As it is a compression task, our parallel design is to **partition the file into equal-size parts**, and assign each thread to work on one partition, then merge thread results to one result.

- **Parallel Read_File & Count_Frequency** (Input Time in below table)

  Each thread counts character frequency of one partition in the whole file, and then merge the result together.

  This is easy to implement since partition can be divided by bytes number. Se can tell each thread read from which byte and end with which byte.

- **Parallel Generate_Compression** (Compression Time in below table)

  Each thread generate the compression of one partition using encoded char set, and output to a local buffer. After they finished, we will concatenate each local output to a global buffer.

  However, it is more difficult than we think early. Since **the bit number of local result may not be the multiple of byte**, which means that a local output may be 30 bits long, and another one is 33 bits long, and our target output is 63 bits long. we can not simply use `memcpy()` or `memmove()` to implement this.

  However, considering that generate_result() costs almost 95% time of total time, we think it's vital to realize its parallelism.

So we **make this problem easier with a little cost on compression ratio**. For a partition which will output 30bits, we will assign 32 bits (4 bytes) to it, and for a 33 bits output, we assign 5 bytes. And then we store the partition bit bound (0:30, 32:67) to the metadata file. So in the process of decode(), we can skip some meaningless bits and get the correct decompressed result.

Although it leads to a bigger compressed file size (9 bytes, the former one is 8 bytes), we get a significant speedup from this.

Deliverables

A executable program is developed and configurable based on the following strategy. The decoding process is not parallelized because it is fast.

Inside the path HuffmanCoding/src, execute `make` first to generate executable files.

The class `HuffmanCompressor` is the main compression algorithm. Class `huffmanBitSet` have some bit operation and is used for store bit code of each character. `main.cpp` input the file, and call the compression function. The usage of `main` is like (you can also execute ./main h for help).

```
./main {e|d} {input_filepath} {output_filepath} {map_filepath} {thread_num}
```

'e' or 'd' means the encode or decode. For encode, `input_filepath` is the original file, `output_file` is the target encoded file path, and `map_filepath` is the target metadata file path.

Notice that `map_filepath` will store the encode map of each character, data size, and decode bit bounds of the original file. We can also move it into the beginning of the decoded file, but in order to make it clear, we make it as an independent file.

The partition number is equal to thread number. The workload of each partition depends on how much characters in the partition, so partitioning by size will make sure that each thread are expected to have a similar work load and reduce the cost due to unbalance.

Result

The records of benchmarking is shown below, with different type of original files.

Correctness is checked by md5sum and diff command.

Compression ratio negligible for sequential and parallel results.

war&peace(32M)					
Algorithm	Thread Num	Input Time (ms)	Compression Time (ms)	Total Time (ms)	**Total Speedup**
Huffman	1	28.332	1658.56	1702.26	1.000
	4	13.6374	667.689	696.701	2.443
	8	11.8753	209.68	236.976	7.183
	16	11.407	107.579	153.547	11.086
	32	11.8537	57.2743	84.5705	20.128
	64	23.626	46.0987	85.2528	19.967
Raw File Bytes:	32022870	Compressed Bytes	17987649	Compression Ratio	1.78
Decode Time	616.528ms				

angular.js(21M)					
Algorithm	Thread Num	Input Time (ms)	Compression Time (ms)	Total Time (ms)	**Total Speedup**
Huffman	1	20.782	1037.930	1087.510	1.000
	4	11.148	271.698	294.180	3.697
	8	8.660	132.590	152.936	7.111
	16	8.530	68.518	88.530	12.284
	32	18.716	37.405	67.559	16.097
	64	14.617	23.218	49.677	21.892
Raw File Bytes:	22046527	Compressed Bytes	13294321	Compression Ratio	1.66
Decode Time	381.781 ms				

Analysis

Since the perfoamce decreases from thread_num=64, which indicates that more threads will hurt the performance, so we did not conduct thread_num=128 test.

- **Speedup**

  At first, with more threads, the speedup increases as expected. However, then the speedup does not have obvious change with the threads number. That's because of the **synchronization of merging results** between threads. In the end of the parallel execution, the program will merge each local results together, it must be sequential since we cannot mix the order of each partition. With more threads, it needs to calls more memcpy(), but each time it just copy a little part of the file.

  The unbalanced workload is not the reason. Since each partition will have the similar number of characters, and that's even for each thread.

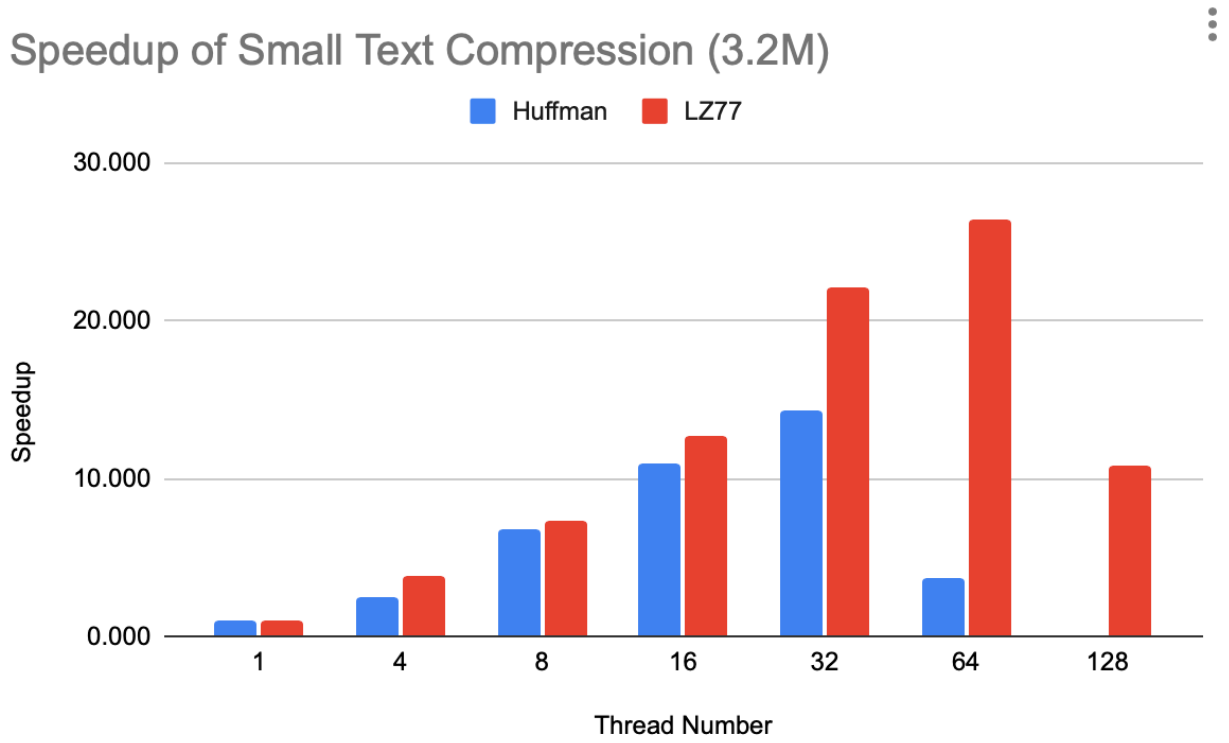
- **Two-part paralleism**

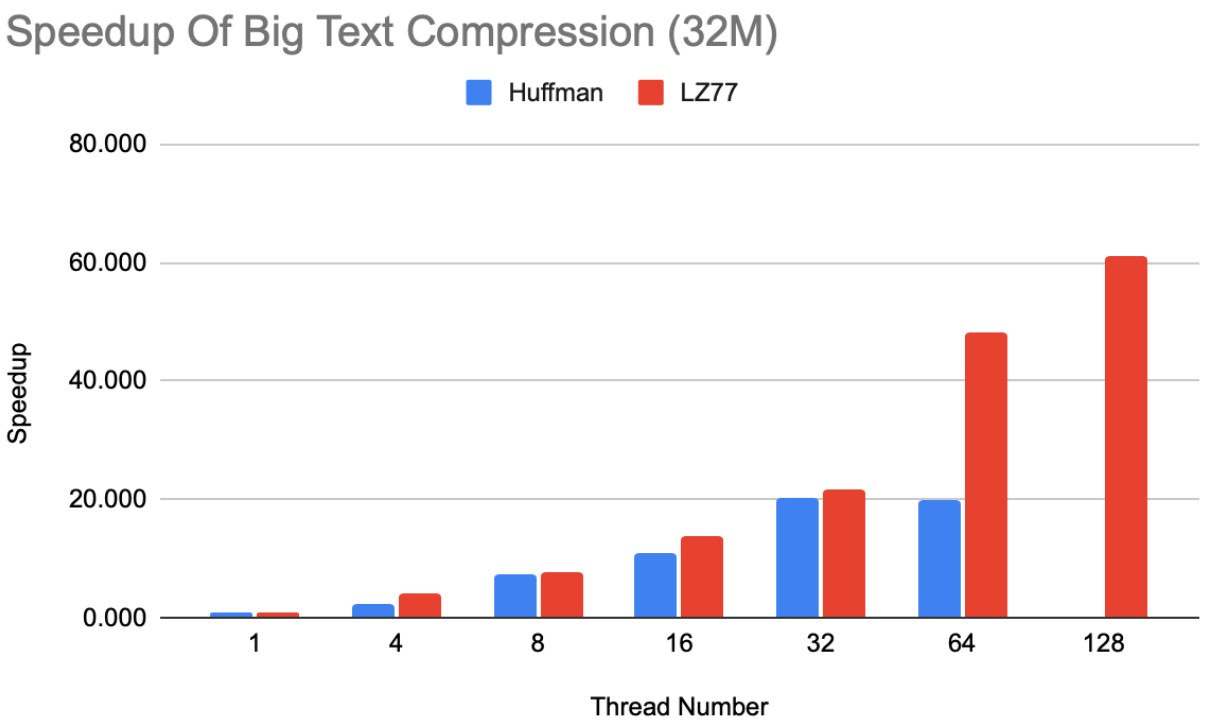
  The implementation of Huffman Coding contains two parts of parallelism, and they have **differnt requirement for threads number**. For the first part, which is read_file and count_frequency (Input Time), when thread number = 4, it has an optimal speedup of 2.2. For the second part, which is generate_compression(Compress Time), when thread number = 8, it has an best speedup of 7.1. When thread number =32, the first part even has some overhead for total performance. In production environment, we can consider **set different threads number for each part** and get an **optimal combination performance**.

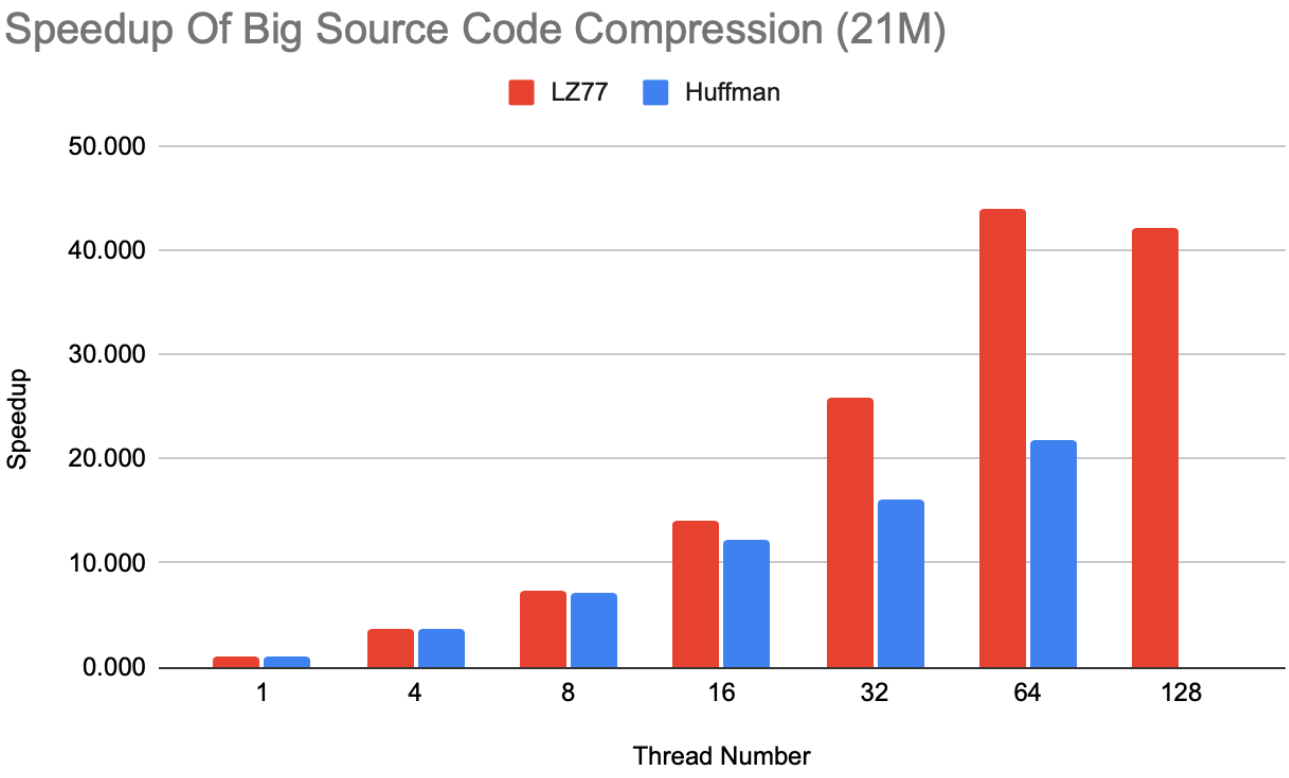
Comparison for Huffman vs. LZ77 and among different traces

Speedup

The speedups for each algorithm on 3 main test traces are shown below:







It could be concluded:

1. LZ77 is genearally more parallelizable than Huffman, probably for its higher ratio of parallelizable part and huffman's overhead of synchronization.

2. Larger files has better speedups than smaller files when number of threads is big, probably for its greater ratio of parallelizable part comparing to the overhead.

3. The performance of parallism is highly trace-dependent. Literature files shows better speedups for LZ77 than source codes, probably for its more even workload distributions. On the other hand, Huffman doesn't suffer from uneven workload distribution.

Compression Ratio

LZ77 shows a much better compression ratio (2.29) than Huffman Coding (1.66) on angular.js, while similar ratio (1.59) with Huffman Coding (1.78) on *war and peace*.

It's because the nature of the different algorithms: LZ77 performs better when content is more duplicated, while Huffman Coding performs better when the byte patterns show greatly different frequencies.

Why not GPU/CUDA?

The reason why we don't implement parallel versions of algorithms on CUDA is clear.

Memory bound and communication overhead

Compression algorithms requires a great amount of memory, that it reads the whole original file into memory and writes backs the whole encoded file.

If apply CUDA-version for the algorithms, the communication between CPU and GPU includes the whole original file and the whole encoded file.

Let's check the throughput of industrial implementations:

Compressor name	Ratio	Compression	Decompress.
zstd 1.4.4 -1	2.884	520 MB/s	1600 MB/s
zlib 1.2.11 -1	2.743	110 MB/s	440 MB/s
brotli 1.0.7 -0	2.701	430 MB/s	470 MB/s
quicklz 1.5.0 -1	2.238	600 MB/s	800 MB/s
lzo1x 2.09 -1	2.106	680 MB/s	950 MB/s
lz4 1.8.3	2.101	800 MB/s	4220 MB/s
snappy 1.1.4	2.073	580 MB/s	2020 MB/s
lzf 3.6 -1	2.077	440 MB/s	930 MB/s

Considering the bandwith of bus, which is probably what limits the communication between CPU and GPU:

PCI-E Version 3.x:

8 GT/s

- **×1:** 985 MB/s
- **×16:** 15.75 GB/s

It's absolutely not reasonable to develop a GPU version of compression algorithms as the bandwith of communication overhead is nearly the throughput of the sequential compression algorithms. Even though our implementation is slow enough to gain speedup with the communication overhead, it's meaningless to do so.

Algorithms are not highly parallelizable (with great number of threads)

As mentioned above, the 2 algorithms are **not** highly parallelizable, especially the Huffman Coding. For both algorithms the ratio of parallelizable part limits the speedups.

What's more, the overhead might outweight the benefits of parallism. For Huffman Coding, it requires extra synchronization (althought the atomic operation is not so costly) to record the frequency of characters, so the overhead increases as the number of threads increase. For LZ77, partitioning breaks character sequences which might lead to a worst compression ratio.

In the tests we conducted, for general file (size 3MB ~ 30MB), the speedup stops increasing or increases very little when the number of threads reach around 32~128 for both algorithms. There is no reason to implement a CUDA-version for compression algorithms as the threads of CUDA are numbers of thousands.

Combination of Two Compressor

Modern compression algorithm like zstd combines these two compressors. According to our analysis, for a compression algorithm **combines** t1 time of Lempel-Ziv family (such as LZ77) encoding and t2 time of Entropy encoding (such as Huffman Coding) of file size m MB, the potential quickest duration is the following:

- Sequential duration:

t1 + t2

- potential best Parallel duration (approximately):

t1/28 + t2/14, when m ~= 3

t1/44 + t2/22, when m ~= 20

T1/61 + t2/22, when m ~= 32

In conclusion, it is worthwhile to parallelize the compression algorithm, and the combination of two compressor will be fitful for more applications.

References

https://en.wikipedia.org/wiki/LZ77_and_LZ78

https://en.wikipedia.org/wiki/Huffman_coding

https://dzone.com/articles/lz-compression-algorithms