Parallel Huffman Coding

Francesco Bozzo
DISI, University of Trento
Trento, Italy
francesco.bozzo@studenti.unitn.it
229312

Michele Yin
DISI, University of Trento
Trento, Italy
michele.yin@studenti.unitn.it
229359

***Abstract*—This report aims to explain the design of a parallel encoding and decoding Huffman algorithm. The application is developed with the C99 programming language, using MPI for multiprocessing and OpenMP for multithreading to scale horizontally with increasing hardware resources. Our tool is both able to exploit multiprocessing to process concurrently multiple files, and to split the Huffman coding of the same file among multiple threads.**

Comparing this tool with other online implementations, we can state that…

***Index Terms*—Huffman, MPI, OpenMP, High Performance Computing**

I. Introduction

Even if the Huffman algorithm is not directly used nowadays, its prefix mechanism is still part of Deflate (PKZIP's algorithm), JPEG, and MP3 compression algorithms

II. The Huffman algorithm

The Huffman algorithm has the objective of finding a more convenient bit representation to store information through lossless compression. Instead of considering groups of eight bits as the way to encode data, the Huffman algorithm uses variable-length sequences of bits with prefixes defined as *alphabet*.

A. Priority Queue

The Huffman algorithm uses a minimum priority queue to build its alphabet efficiently. This specific data structure ensures logarithmic insertion and deletion time with respect to its size. We implemented the minimum priority queue by using a minimum heap. Practically speaking, to implement the min-heap tree, we used a standard C array ensuring that the min-heap property still holds at every insertion and deletion:

$$A[i] \leq A[l(i)], A[i] \leq A[r(i)] \tag{1}$$

where $A[i]$, $A[l(i)]$, $andA[r(i)]$ are respectively a node, its left child, and its right child in a min-heap tree.

To implement the priority queue also a parallel approach has been considered [1], but since it contains only 256 elements (1 byte), we did not consider it too important in terms of performance.

B. Encoding

The Huffman encoding procedure makes use of a minimum priority queue to build its alphabet efficiently. The idea is to build a tree similar to Figure 1 that defines all the variable-length prefix sequences of bits: these sequences are represented by the path from the tree root to a leaf. Using a greedy approach, the Huffman encoding ensures that the less frequent a byte is in a file, the more probable is to have a longer Huffman representation, which means that his path from the root to its specific leaf is longer.

Once having computed the frequencies for each one of the 256 different bytes in a file, the Huffman algorithm populates a min priority queue with Huffman tree nodes storing the byte value and its frequency. Successively, it removes the least two frequent bytes from the queue, creates a new node with children the two extracted nodes, assigns to it a dummy character and the sum of the frequencies of its two children, and inserts it into the min priority queue. After $n-1$ iterations, the last node is the root of the Huffman tree [2]. Algorithm 1 explains in the detail this procedure.

Algorithm 1: Build the Huffman tree

```
// Populate the min priority queue with characters and their frequencies
for $i = 1$ to $n - 1$ do
    Q.insert(f[i], Tree(f[i], c[i]))
// Repeat until the queue has only a single element left
for $i = 1$ to $n - 1$ do
    // Get the two least frequent nodes
    z1 = Q.deleteMin()
    z2 = Q.deleteMin()
    // Create an inner tree node and insert it into the queue
    z = Tree(z1.f + z2.f, null)
    z.left = z1
    z.right = z2
    Q.insert(z.f, z)
// The last element in the queue is the root of the Huffman tree
return Q.deleteMin()
```

Once built the Huffman tree, it is possible to generate the

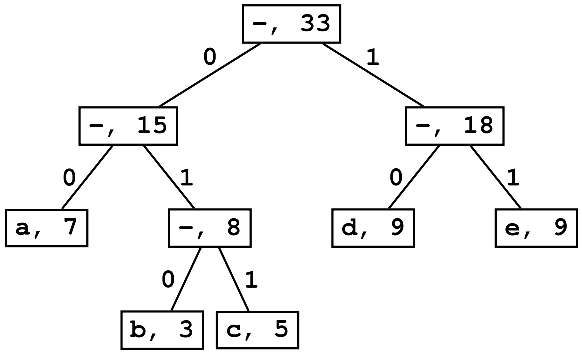

Fig. 1: An example of Huffman tree.

Huffman alphabet visiting the tree using a DFS algorithm, assigning 0 to each left-child traverse and 1 for the right one as presented in Algorithm 2. Finally, it is possible to compress the file by creating a stream of bits corresponding to its content using the Huffman alphabet.

Algorithm 2: Encode using Huffman tree

```
while not eof() do
    bit = read()
    if bit == 0 then
        encode(node.left)
    else
        encode(node.right)
    if node is leaf then
        return node.value
```

C. Decoding

Once having the encoded file and the Huffman tree, it is straightforward to decompress the file to its original shape. The prefix alphabet ensures to have that it is possible to visit the tree using a DFS approach and get a unique decoded version of the file using an approach similar to Algorithm 2.

III. Serial Huffman implementation

To be able to implement the Huffman encoding and decoding algorithm, several things have been taken into account.

- The byte frequencies computed during the encoding process are saved in the compressed file. In this way, the decoding procedure can easily rebuild the Huffman tree.
- Both the encoding and decoding procedures make use of buffers to improve I/O performance: the streams of bytes are first written inside the buffer and then saved on the disk.
- The encoding procedure works with chunks of 4096 bytes. This ensures that the tool can handle even large files when dealing with bit buffers. The decoding procedure deals with chunks of maximum size of $4096 * 32$ bytes, since the maximum length of a Huffman tree traverse from root to leaf is of 256 bits, which is 32 times a byte. As explained later, dealing with chunks makes handy the parallel implementation of the encoding and decoding algorithms.
- In the encoded file, the chunk offsets are saved at the end of the file: this ensures that the compressed stream of prefixed sequences of bits is still meaningful.

IV. Parallel Huffman implementation

To parallelize the serial Huffman algorithm over threads and processes, we decided to follow this simple yet effective structure:

- Multiple processes should handle groups of file and folders separately.
- Multiple threads of the same process should work on different chunks of the same file in parallel.

Specifically, there are multiple reasons behind our parallel architectural design for this project. Here follows a list of the main ones:

- In most operating systems a file is a resource that the OS gives to a single process to avoid race conditions. We wanted to follow a similar design philosophy.
- Most operating systems allow multiple processes to open the same file in reading mode, but only few allow to open the same file in writing mode on multiple processes. This is because that leads to potentially concurrency and data integrity issues. By ensuring to have only a single process that open a specific file, we avoid all these issues.
- Because threads of the same process share the address space, we can avoid the expensive data transfer across processes. When data is read or written to file, there is no need to transfer data between threads.

In the next section, we start by explaining the basic multi-threading operations on a single file. Later on, multiprocessing with multiple files is covered.

A. Multithreading

The parallel algorithm for a single file follows the exact same procedure as for the serial version, with just a few key differences to allow multithreading.

Suppose we are dealing with m threads and a single input file. Here follows a common procedure used for both encoding and decoding, with very small differences.

1) A single file is divided into chunks of a fixed size. When that is done, the single process forks and creates m threads.
2) A single thread (may be the main one, may not) then reads m chunks in a shared memory buffer.
3) Each one of the m threads works in parallel on the processing of its assigned chunk using a buffer. Although all the chunks are in shared memory buffer, each thread is assigned only to a single memory section: in this way there is no need to care about racing conditions when writing in the buffer. This is done by creating a `unsigned char buffer[m][buffer_size]`.

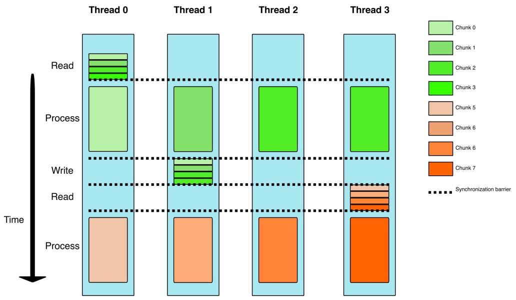

Fig. 2: Simple schema for processing a single file with multiple threads.

4) When all threads are done computing, either the encoding or decoding of their chunks, a single thread writes the processed chunks to the output file. Again, since all threads share a memory space there is no need to perform data transfer.

Moreover, our implementation also parallelize the counting of occurrences of a byte in the input file, since this is required in order to create the Huffman tree. This is done only during the encoding phase, as for the decoding the occurences are no longer required because the huffman tree is already saved in the encoded data. The procedure is very similar to the one just described, with the key difference that there is no output file to write, but only byte occurrences stored in an in-memory array unique for each thread to avoid concurrency. When the counting is done, all threads join their counts into a single array.

Figure 2 shows a schema of the parallel workflow for processing a single file.

B. Multiprocessing

Additionally, we wanted our tool to be able to process multiple files and folders: for this use case, we exploit multiprocessing.

To achieve this, we devised a simple algorithm to distribute files across multiple processes, so that each process can work on its own list of files, independently to others. In this way, we can minimize communication and therefore latency and other slowdowns between the different components of the applications. Suppose we are dealing with n process, m threads, and an input folder.

1) First, the process with rank 0 (the main process) crawls all the files in the input directory recursively in all the subfolders.
2) Then the rank 0 process opens all the files and reads their size.
3) Once all files and their respective sizes are known, the main process creates a min priority queue where each item represents a process and its priority is the size of files that have been assigned to it.
4) Process 0 can iteratively insert files in the priority queue and updates the priority of each process with the cumulative file size. The main idea is that we can ensure an equal work division among different processes.
5) When all this is done, the main process sends the list of files to each process and then each starts to work independently on its own list of files.

C. Implementation details and other notes

Here follows a list of further comments and design choices for our specific Huffman implementation.

- We decided on an alphabet of 256 for the Huffman coding because this way we can encode each byte of the input data into a different Huffman code. As C language does not allow to address data at lower resolution than a byte, lowering the alphabet would not result in any perfomance benefit. We also considered encoding multiple bytes into a single huffman code, but that results in worse compression as the resulting huffman tree would have more leaves.
- We found that a chunk size of 4096 bytes had the best I/O times. This is probably due to the fact that 4096 B is the size of a page in most Linux based operating systems. The last chunk of a file may be smaller. Reading any other size at the time had resulted in significantly worse I/O times (both increasing and decreasing the chunk size).
- In general, all threads will finish processing their chunks at around the same time, but in some cases there are threads that can take longer. This is due to the very unlucky case where a whole chunk contains exclusively very infrequent bytes of data. Because the bytes are very infrequent, each one is encoded in extremely long sequences, up to 256 bits, resulting in a bigger encoded chunk than the original data. With our Huffman alphabet set to a single byte, the worst case results in having a compressed chunk being 32 times bigger than decompressed one.
- We tried to parallelize the I/O, by having each thread read its own chunk. We found that the standard file descriptor provided by C have a lock to guarantee thread safety. Unfortunately, this lock slows down the reads significantly. We also tried using multiple file descriptors to circumvent this limitation, however we found no improvement over a single-thread sequential read. This is likely because the O.S. schedules I/O requests and serves them one at the time, resulting in an impossibility of having truly parallel I/O.
- We also tested an architecture where we had two dedicated threads to I/O, one for reading and one for writing chunks. The idea is that whenever a chunk is processed, the I/O threads would immediately write the processed chunk to the output file and similarly a new chunk would be read from the input file (very similar to a queue of jobs). Theoretically this would allow to parallelize I/O and computation operations: in this scenario we avoided concurrency problems by synchronizing the I/O with locks. In later analysis we call this architecture "Locks" because of the synchronization method we are using.

- Another detail we noticed on the implementation with two dedicated threads for I/O, is that if there are not enough cores on the CPU or if the operating system decides to allocate all the threads on a single processor, the encoding and decoding times are greatly affected by the scheduler. This is because it might be that the O.S. gives priority to threads that are waiting (for example the writer cannot write any block until the first has finished, even if all others have finished). In the worst case scenario, the multithreading architecture could be even slower that the serial one.

V. Performance and Benchmarking

In this section we are going to discuss the performance evaluation of our algorithm. We start from evaluating some other implementations of Huffman algorithm and then focus on the analysis of our own implementation.

We include in this section some graphs and statistics, but all detailed results are included in the appendix at the end this report.

A. Setup

All the tests have been performed on the HPC2 cluster of the University of Trento. To reduce as much as possible the I/O timings, which are crucial of our application, jobs were run in a very specific setting:

- Threads are allocated in the same node and, if possible, on the same socket to guarantee fast communication between them. This has been achieved by using the MPI option `--map-by`.
- Processes are allocated in different nodes, by using the PBS directive `place=scatter:excl`. This because in our application different processes do not communicate very often between each other, but they just split their work and use the file-system.

The provided timings are obtained by averaging the result of three runs with the exact same configurations: this helps to minimize the effect random variance between runs due to potential hardware congestion. Each job has been submitted to the cluster individually, waiting for the previous to finish preventing hiccups due to multiple instances of the program trying to access the same file-system resource, which we noticed to greatly affect I/O times.

We extensively verified the correctness of our algorithm by encoding and then decoding a file, then comparing the decoded result to the original file by using the `diff` command. Moreover, Valgrind has been used to spot any memory leaks that could cause runtime errors.

B. Datasets

The evaluation dataset has been randomly generated by using a simple C program: we created text files with character frequencies that follow the occurrences of the letters in English language. Although for benchmarking we are only using ASCII characters, please note that our algorithm reads bytes of data and can therefore work with any file. The dataset is composed by files of increasing size: 1 MiB, 5 MiB, 10 MiB, 50 MiB, 100 MiB, 500 MiB, 1 GiB, 5 GiB, and 10 GiB. This should be enough range to understand how different algorithms scale with increasing file sizes.

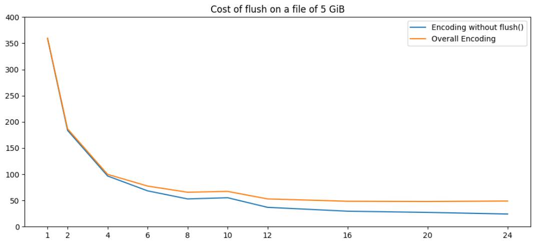

Fig. 3: Encoding times with and without the flush()

Secondly, because our algorithm also works with many files at once, we chose some GitHub repository as benchmark: Numpy, PyTorch and Linux kernel as benchmarks. They are respectively 30MiB, 200MiB and 1.3 GiB of total size, with about $\sim$ 2.000, $\sim$ 12.000, and $\sim$ 80.000 files, which on average are each 16MiB in size.

Considering the dataset sizes and amount testing we have done, we estimate to have accessed the disk (both read and writes) for about $\sim$ 6 TiB of data for our whole benchmark analysis.

C. Other works

D. Multithreading evaluation

In this section we evaluate our algorithm on single files, by using multithreading and testing our parallelization performances with an increasing number of threads.

With small files (i.e. less than a few MiBs) the algorithm is actually slower with more threads. This is likely to be the case because the data is not large enough to offset the cost of creating multiple threads. Considering the cost parallel processing, we highly suggest not doing parallelization for small files.

However, when the data is large enough our multithreaded approach scales up. With the largest file (10 GiB) our algorithm has an efficiency of 93% when tested with 4 threads. This number steadily decreases with the number of threads, and we hit 32% with 24 threads.

Please note that our tool is heavily limited by the cluster I/O, since we can not make it parallel: especially writings to disk require a considerable amount of time. As a proof of this, if we consider only the processing section of our algorithm, and ignore the time required by the `fflush()` operation before a `fclose()`, we see significantly decreased times. Although we start from similar efficiency at low threading, we achieve a 54% efficiency with 24 threads in this case: if we consider only CPU time, which does not count the time spent waiting in I/O, we get very promising results.

We also noticed worse times in the decoding procedure with respect to encoding. This is probably due to the fact that the tool is reading fixed chunks of 4096 bytes in encoding, but during decoding the chunk can have a variable length.

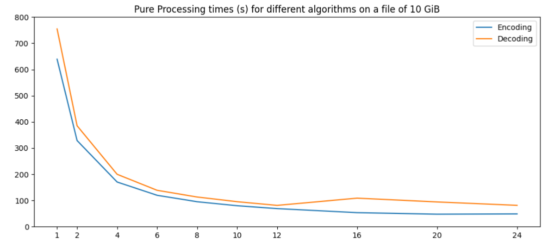

Fig. 4: Encoding vs Decoding times

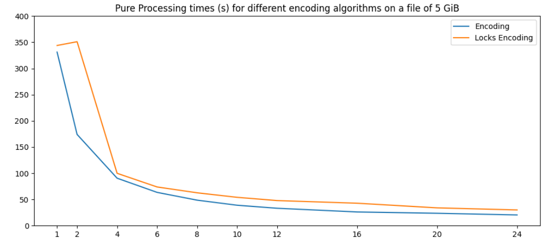

Fig. 5: Barrier vs Lock based synchronization performance

As already discussed previously, we also tested a version on a lock-based synchronization instead of barriers. Although theoretically we should get better results, in practice we obtain almost equal performances across the board, with only the largest files having some significant difference. This difference is greater in decoding than in encoding.

E. Multiprocessing

We tested on different folders as benchmark the parallelization performances of our algorithm. Theoretically, our algorithm should scale very well with increasing number of processes and files, because once the main process sends the list of files to encode to each process there is no further communication between different processes. However, we find that real world results are different. While we don't expect much increasing in the performance by the number of threads, because the files are on average very small, we should get better results by using many processes. In reality our testing shows that this is not the case. Our efficiencies are extremely low. We believe the main limitation of our algorithm is the I/O of the cluster. When tested on a local machine (MacBook A2442) with one thread and 1,2,4 processes, we find that our algorithm does indeed scale with the number of processes. Moreover, we were not able to test Linux on the cluster due to extreme I/O times.

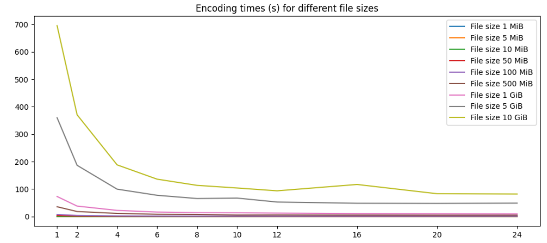

Fig. 6: Encoding times

F. Memory, storage and other considerations

Our memory requirements don't scale upwards with the size of the files we need to encode, because at any given time, the most we only store is the size of two buffers, each is a `unsigned char buffer[num_threads][4096]`. These buffers are emptied at each I/O cycle, therefore leading to a high memory efficiency. Only when processing multiple files our memory requirements scale up with the number of files, as each process needs to store information about the files is assigned to process.

We also find that we achieve on average 48% compression rate of our synthetic data. Instead, on Linux kernel, because contains more varied data which consists of mostly code, we achieve a lower 62% compression rate. Testing on other already compressed data as `.zip` or `AV1` resulted in a compression rate of 100%.

VI. Conclusions

To conclude.

References

[1] G. S. Brodal, J. L. Träff, and C. D. Zaroliagis, "A parallel priority queue with constant time operations," *Journal of Parallel and Distributed Computing*, vol. 49, no. 1, pp. 4–21, 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731598914253

[2] A. Bertossi and A. Montresor, *Algoritmi e strutture di dati*. CittàStudi, 2010. [Online]. Available: https://books.google.es/books?id=WKWLSQAACAAJ

VII. Appendix

A. Encoding

TABLE I: Overall encoding times

Threads File sizes	1	2	4	6	8	10	12	16	20	24
1 MiB	0.1084	0.1029	0.0826	0.0905	0.0718	0.1100	0.1338	0.2110	0.1705	0.2531
5 MiB	0.4073	0.2008	0.1255	0.1037	0.0970	0.0786	0.0789	0.0767	0.0906	0.0883
10 MiB	0.7611	0.4737	0.2658	0.2084	0.1804	0.1861	0.2204	0.3353	0.2432	0.3930
50 MiB	3.9241	2.1378	1.2788	1.0203	0.8575	0.7589	0.7773	0.6583	0.7523	0.9954
100 MiB	7.7270	4.0680	2.4028	1.8596	1.5545	1.4324	1.2994	1.2468	1.2308	1.5375
500 MiB	35.7753	18.3831	11.4959	8.3511	7.5589	5.7548	6.0857	5.5698	5.1431	5.0972
1 GiB	73.1120	38.1679	22.5066	16.3364	14.2131	13.7565	12.6196	10.8261	10.3345	9.7426
5 GiB	359.8495	186.6294	99.8038	77.4620	65.6555	67.3056	52.9090	48.4594	48.0126	48.8973
10 GiB	694.9750	370.5490	188.2278	136.1719	113.5878	104.0254	93.7227	116.8249	83.2084	82.0529

TABLE II: Pure encoding times

Threads File sizes	1	2	4	6	8	10	12	16	20	24
1 MiB	0.0637	0.0338	0.0184	0.0128	0.0100	0.0085	0.0388	0.0938	0.0351	0.1032
5 MiB	0.2980	0.1522	0.0782	0.0537	0.0413	0.0341	0.0296	0.0243	0.0207	0.0264
10 MiB	0.6547	0.3523	0.1845	0.1254	0.0977	0.0812	0.0730	0.1533	0.0570	0.1505
50 MiB	3.3027	1.7545	0.9147	0.6244	0.4884	0.4326	0.3432	0.2658	0.2953	0.3141
100 MiB	6.7211	3.4461	1.8267	1.2448	0.9903	0.8153	0.6873	0.5415	0.4856	0.6148
500 MiB	32.3825	16.6046	8.6817	6.2457	4.8624	4.0588	3.4132	2.6616	2.3853	2.0639
1 GiB	67.1756	34.0970	17.7353	12.6548	9.8790	8.2365	7.0042	5.4307	4.8915	4.2098
5 GiB	331.2526	174.0889	90.5618	63.4905	48.6453	38.9458	33.1852	26.1005	23.6226	20.4274
10 GiB	639.3369	328.5801	170.4086	119.6824	95.1232	79.8054	68.8753	53.3415	47.5727	48.6436

TABLE III: Overall encoding efficiency

Threads File sizes	1	2	4	6	8	10	12	16	20	24
1 MiB	1.0	0.5272	0.3283	0.1997	0.1889	0.0986	0.0676	0.0321	0.0318	0.0179
5 MiB	1.0	1.0144	0.8111	0.6544	0.5250	0.5182	0.4302	0.3317	0.2247	0.1922
10 MiB	1.0	0.8034	0.7158	0.6086	0.5273	0.4090	0.2878	0.1419	0.1565	0.0807
50 MiB	1.0	0.9178	0.7671	0.6410	0.5721	0.5171	0.4207	0.3726	0.2608	0.1643
100 MiB	1.0	0.9497	0.8040	0.6925	0.6213	0.5394	0.4956	0.3873	0.3139	0.2094
500 MiB	1.0	0.9731	0.7780	0.7140	0.5916	0.6217	0.4899	0.4014	0.3478	0.2924
1 GiB	1.0	0.9578	0.8121	0.7459	0.6430	0.5315	0.4828	0.4221	0.3537	0.3127
5 GiB	1.0	0.9641	0.9014	0.7742	0.6851	0.5346	0.5668	0.4641	0.3747	0.3066
10 GiB	1.0	0.9378	0.9231	0.8506	0.7648	0.6681	0.6179	0.3718	0.4176	0.3529

TABLE IV: Pure encoding efficiency

Threads File sizes	1	2	4	6	8	10	12	16	20	24
1 MiB	1.0	0.9432	0.8677	0.8304	0.7960	0.7472	0.1370	0.0424	0.0908	0.0257
5 MiB	1.0	0.9788	0.9521	0.9253	0.9025	0.8729	0.8402	0.7654	0.7194	0.4697
10 MiB	1.0	0.9290	0.8872	0.8703	0.8373	0.8061	0.7476	0.2668	0.5743	0.1813
50 MiB	1.0	0.9412	0.9027	0.8816	0.8454	0.7635	0.8021	0.7767	0.5593	0.4381
100 MiB	1.0	0.9752	0.9198	0.8999	0.8484	0.8244	0.8149	0.7757	0.6920	0.4555
500 MiB	1.0	0.9751	0.9325	0.8641	0.8325	0.7978	0.7906	0.7604	0.6788	0.6538
1 GiB	1.0	0.9851	0.9469	0.8847	0.8500	0.8156	0.7992	0.7731	0.6867	0.6649
5 GiB	1.0	0.9514	0.9144	0.8696	0.8512	0.8505	0.8318	0.7932	0.7011	0.6757
10 GiB	1.0	0.9729	0.9379	0.8903	0.8401	0.8011	0.7735	0.7491	0.6720	0.5476

B. *Encoding with Locks*

TABLE V: Overall encoding times

Threads File sizes	1	2	4	6	8	10	12	16	20	24
1 MiB	0.0880	0.0837	0.0732	0.0654	0.0724	0.0724	0.0672	0.0667	0.0752	0.0712
5 MiB	0.4246	0.4062	0.2753	0.1725	0.1674	0.1642	0.1479	0.1389	0.1387	0.1362
10 MiB	0.7128	0.6782	0.2417	0.1939	0.1762	0.1471	0.1676	0.1286	0.1403	0.1570
50 MiB	3.5854	3.3710	1.1411	0.8380	0.7478	0.6985	0.9610	0.9681	0.7327	0.8949
100 MiB	6.8047	6.6007	2.1585	1.6222	1.3589	1.2004	1.0981	0.9736	0.8877	0.8418
500 MiB	33.4643	31.9953	10.5641	7.8537	6.6834	6.0406	5.5178	5.9307	4.5767	4.2484
1 GiB	68.4891	65.9418	21.1728	16.4254	13.6481	12.0767	10.9664	9.7954	9.0356	8.3424
5 GiB	367.2282	370.8502	110.2986	84.1276	75.3192	66.4084	59.6663	59.4788	51.1808	48.9043
10 GiB	676.5289	642.9220	187.6202	135.9907	123.8217	109.7696	96.6426	93.2259	90.8253	88.1542

TABLE VI: Pure encoding times

Threads File sizes	1	2	4	6	8	10	12	16	20	24
1 MiB	0.0612	0.0621	0.0360	0.0278	0.0303	0.0216	0.0244	0.0239	0.0301	0.0158
5 MiB	0.3611	0.3633	0.2093	0.1092	0.1031	0.0923	0.0792	0.0657	0.0530	0.0501
10 MiB	0.6167	0.6074	0.1689	0.1167	0.0905	0.0756	0.0784	0.0540	0.0529	0.0590
50 MiB	3.1314	3.0569	0.8470	0.5946	0.4641	0.4157	0.6712	0.6516	0.4349	0.4997
100 MiB	6.0792	6.0371	1.6635	1.1509	0.8906	0.7352	0.6421	0.5210	0.4383	0.3862
500 MiB	31.0011	30.4712	8.2873	5.8097	4.5192	3.7783	3.2265	2.6096	2.2273	1.9458
1 GiB	62.7481	62.3551	17.2300	11.9387	9.1530	7.5818	6.5052	5.3627	4.6125	4.1011
5 GiB	343.8756	351.0666	99.9091	73.8247	62.6944	54.0495	47.8488	42.7941	33.9437	29.9516
10 GiB	629.4486	623.7099	172.8286	120.5519	94.6229	78.7281	68.0250	55.2154	46.7232	41.4739

TABLE VII: Overall encoding efficiency

Threads File sizes	1	2	4	6	8	10	12	16	20	24
1 MiB	1.0	0.5258	0.3004	0.2242	0.1519	0.1216	0.1091	0.0825	0.0585	0.0515
5 MiB	1.0	0.5227	0.3856	0.4104	0.3171	0.2585	0.2392	0.1911	0.1531	0.1299
10 MiB	1.0	0.5255	0.7371	0.6127	0.5057	0.4845	0.3543	0.3465	0.2541	0.1892
50 MiB	1.0	0.5318	0.7855	0.7130	0.5994	0.5133	0.3109	0.2315	0.2447	0.1669
100 MiB	1.0	0.5155	0.7881	0.6991	0.6260	0.5669	0.5164	0.4368	0.3833	0.3368
500 MiB	1.0	0.5230	0.7919	0.7102	0.6259	0.5540	0.5054	0.3527	0.3656	0.3282
1 GiB	1.0	0.5193	0.8087	0.6949	0.6273	0.5671	0.5204	0.4370	0.3790	0.3421
5 GiB	1.0	0.4951	0.8323	0.7275	0.6095	0.5530	0.5129	0.3859	0.3588	0.3129
10 GiB	1.0	0.5261	0.9015	0.8291	0.6830	0.6163	0.5834	0.4536	0.3724	0.3198

TABLE VIII: Pure encoding efficiency

Threads File sizes	1	2	4	6	8	10	12	16	20	24
1 MiB	1.0	0.4928	0.4246	0.3672	0.2522	0.2831	0.2086	0.1598	0.1015	0.1611
5 MiB	1.0	0.4969	0.4312	0.5511	0.4378	0.3912	0.3801	0.3437	0.3409	0.3001
10 MiB	1.0	0.5077	0.9129	0.8805	0.8520	0.8159	0.6554	0.7144	0.5827	0.4356
50 MiB	1.0	0.5122	0.9243	0.8778	0.8433	0.7532	0.3888	0.3003	0.3601	0.2611
100 MiB	1.0	0.5035	0.9136	0.8804	0.8532	0.8268	0.7890	0.7292	0.6934	0.6560
500 MiB	1.0	0.5087	0.9352	0.8893	0.8575	0.8205	0.8007	0.7425	0.6959	0.6638
1 GiB	1.0	0.5032	0.9104	0.8760	0.8569	0.8276	0.8038	0.7313	0.6802	0.6375
5 GiB	1.0	0.4898	0.8605	0.7763	0.6856	0.6362	0.5989	0.5022	0.5065	0.4784
10 GiB	1.0	0.5046	0.9105	0.8702	0.8315	0.7995	0.7711	0.7125	0.6736	0.6324

C. Decoding

TABLE IX: Overall decoding times

Threads File sizes	1	2	4	6	8	10	12	16	20	24
1 MiB	0.1304	0.0907	0.0793	0.0986	0.0912	0.0975	0.1179	0.1152	0.1185	0.1039
5 MiB	0.4219	0.2527	0.1772	0.1663	0.1411	0.1563	0.1507	0.1395	0.1296	0.1214
10 MiB	0.8712	0.5173	0.3839	0.2955	0.2931	0.4331	0.3716	0.3557	0.3683	0.3546
50 MiB	4.1082	2.2768	1.5108	1.2518	1.2762	1.3824	1.4124	1.4088	1.4343	1.3277
100 MiB	7.9605	4.5073	2.9151	2.2752	1.8842	2.4183	2.4003	2.4222	2.3432	2.2511
500 MiB	37.6189	22.5113	13.5012	11.3339	10.0352	9.3255	9.6405	11.1822	11.0692	10.0214
1 GiB	76.6794	43.2517	27.5849	23.2715	18.5904	17.8240	18.5319	22.0271	19.8127	17.1276
5 GiB	361.8713	193.1122	108.8124	96.7901	81.0661	84.8969	79.1422	73.7254	79.3745	75.9761
10 GiB	759.0719	391.2016	212.3814	156.3964	135.8435	135.4010	134.0983	137.9998	134.0065	132.2553

TABLE X: Pure decoding times

Threads File sizes	1	2	4	6	8	10	12	16	20	24
1 MiB	0.0833	0.0579	0.0519	0.0737	0.0693	0.0712	0.0920	0.0903	0.0924	0.0761
5 MiB	0.3397	0.1883	0.1231	0.1014	0.0843	0.0942	0.0893	0.0789	0.0729	0.0623
10 MiB	0.7516	0.4136	0.2667	0.1912	0.1905	0.2648	0.2672	0.2459	0.2821	0.2490
50 MiB	3.6338	1.8884	1.0578	0.7449	0.6699	0.9762	0.9542	0.9429	0.9658	0.8575
100 MiB	7.1318	3.6134	2.0121	1.3884	1.1225	1.5401	1.5201	1.5413	1.4775	1.4684
500 MiB	36.3947	18.5941	9.8736	6.9800	5.4581	4.7071	5.7395	6.9697	6.4991	5.4733
1 GiB	74.7637	38.4079	19.7644	13.8396	10.9082	10.2419	9.2313	12.5040	12.0604	9.8245
5 GiB	359.8834	188.8855	97.7879	82.3554	56.9994	59.9356	50.8868	39.2907	48.5617	48.8765
10 GiB	754.8416	385.0213	199.8499	138.7388	113.1048	95.2621	81.1119	108.7269	94.3171	81.2615

TABLE XI: Overall decoding efficiency

Threads File sizes	1	2	4	6	8	10	12	16	20	24
1 MiB	1.0	0.7186	0.4108	0.2204	0.1787	0.1338	0.0921	0.0707	0.0550	0.0523
5 MiB	1.0	0.8349	0.5953	0.4228	0.3739	0.2700	0.2334	0.1890	0.1628	0.1448
10 MiB	1.0	0.8421	0.5673	0.4913	0.3715	0.2012	0.1954	0.1531	0.1183	0.1024
50 MiB	1.0	0.9022	0.6798	0.5469	0.4024	0.2972	0.2424	0.1823	0.1432	0.1289
100 MiB	1.0	0.8831	0.6827	0.5831	0.5281	0.3292	0.2764	0.2054	0.1699	0.1473
500 MiB	1.0	0.8356	0.6966	0.5532	0.4686	0.4034	0.3252	0.2103	0.1699	0.1564
1 GiB	1.0	0.8864	0.6949	0.5492	0.5156	0.4302	0.3448	0.2176	0.1935	0.1865
5 GiB	1.0	0.9369	0.8314	0.6231	0.5580	0.4262	0.3810	0.3068	0.2280	0.1985
10 GiB	1.0	0.9702	0.8935	0.8089	0.6985	0.5606	0.4717	0.3438	0.2832	0.2391

TABLE XII: Pure decoding efficiency

Threads File sizes	1	2	4	6	8	10	12	16	20	24
1 MiB	1.0	0.7196	0.4013	0.1884	0.1504	0.1170	0.0755	0.0577	0.0451	0.0456
5 MiB	1.0	0.9021	0.6897	0.5580	0.5038	0.3606	0.3168	0.2689	0.2330	0.2271
10 MiB	1.0	0.9087	0.7046	0.6550	0.4931	0.2839	0.2344	0.1911	0.1332	0.1258
50 MiB	1.0	0.9621	0.8588	0.8131	0.6781	0.3722	0.3174	0.2409	0.1881	0.1766
100 MiB	1.0	0.9869	0.8861	0.8562	0.7942	0.4631	0.3910	0.2892	0.2413	0.2024
500 MiB	1.0	0.9787	0.9215	0.8690	0.8335	0.7732	0.5284	0.3264	0.2800	0.2771
1 GiB	1.0	0.9733	0.9457	0.9004	0.8567	0.7300	0.6749	0.3737	0.3100	0.3171
5 GiB	1.0	0.9526	0.9201	0.7283	0.7892	0.6004	0.5894	0.5725	0.3705	0.3068
10 GiB	1.0	0.9803	0.9443	0.9068	0.8342	0.7924	0.7755	0.4339	0.4002	0.3870

D. Decoding with Locks

TABLE XIII: Overall decoding times

Threads File sizes	1	2	4	6	8	10	12	16	20	24
1 MiB	0.2719	0.2116	0.1162	0.1006	0.0699	0.0677	0.0656	0.0695	0.0748	0.0928
5 MiB	0.6351	0.9155	1.2650	0.6553	0.6837	0.2889	0.4825	0.2982	0.3296	0.1795
10 MiB	0.8607	0.5007	0.3328	0.2710	0.2687	0.2580	0.2364	0.2169	0.1899	0.1830
50 MiB	3.9914	2.2478	1.4618	1.1756	1.1596	1.0884	1.0173	0.9127	0.8369	0.7747
100 MiB	7.6155	33.6453	15.2498	11.6282	13.0181	10.7612	9.1921	3.6613	5.6912	4.4747
500 MiB	37.0819	20.4215	13.2360	11.3439	10.7738	10.4405	9.7670	8.6608	7.3871	7.0508
1 GiB	73.8554	41.1849	26.9294	23.0511	17.9771	18.7808	19.7312	17.8374	16.1711	15.4367
5 GiB	352.3856	475.0869	265.8940	204.6614	156.2271	139.8945	127.6489	109.2454	100.9185	89.8899
10 GiB	714.9067	375.6862	202.7149	190.6061	435.2239	145.6734	133.3188	122.7745	117.5286	115.2620

TABLE XIV: Pure decoding times

Threads File sizes	1	2	4	6	8	10	12	16	20	24
1 MiB	0.1162	0.0645	0.0563	0.0555	0.0407	0.0408	0.0379	0.0413	0.0382	0.0351
5 MiB	0.5650	0.8586	1.2021	0.5983	0.6266	0.2339	0.4245	0.2443	0.2693	0.1130
10 MiB	0.7149	0.3986	0.2338	0.1743	0.1690	0.1591	0.1363	0.1187	0.0949	0.0849
50 MiB	3.5310	1.8481	1.0339	0.7423	0.7299	0.6557	0.6202	0.4858	0.4169	0.3519
100 MiB	6.8744	33.3659	14.8835	10.7860	12.3773	9.9066	8.3482	2.8794	5.0465	3.6955
500 MiB	35.4497	18.2381	10.0089	7.1791	6.7919	6.3173	5.6216	4.6675	3.8306	3.2797
1 GiB	72.1138	36.9260	20.1134	14.5033	11.5997	12.6916	11.3021	9.2808	7.6540	6.4776
5 GiB	349.8904	471.3404	257.7142	197.0613	140.1541	124.8532	110.6343	91.4010	74.5820	62.5393
10 GiB	711.9846	370,5192	191.8006	182.3177	421.9179	126.6782	113.4824	92.0815	76.3088	64.4831

TABLE XV: Overall decoding efficiency

Threads File sizes	1	2	4	6	8	10	12	16	20	24
1 MiB	1.0	0.6424	0.5848	0.4504	0.4859	0.4018	0.3456	0.2446	0.1818	0.1220
5 MiB	1.0	0.3469	0.1255	0.1615	0.1161	0.2199	0.1097	0.1331	0.0963	0.1474
10 MiB	1.0	0.8595	0.6466	0.5293	0.4005	0.3336	0.3033	0.2480	0.2266	0.1959
50 MiB	1.0	0.8878	0.6826	0.5659	0.4303	0.3667	0.3269	0.2733	0.2385	0.2147
100 MiB	1.0	0.1132	0.1248	0.1092	0.0731	0.0708	0.0690	0.1300	0.0669	0.0709
500 MiB	1.0	0.9079	0.7004	0.5448	0.4302	0.3552	0.3164	0.2676	0.2510	0.2191
1 GiB	1.0	0.8966	0.6856	0.5340	0.5135	0.3932	0.3119	0.2588	0.2284	0.1993
5 GiB	1.0	0.3709	0.3313	0.2870	0.2819	0.2519	0.2300	0.2016	0.1746	0.1633
10 GiB	1.0	0.9506	0.8817	0.6251	0.2053	0.4908	0.4469	0.3639	0.3041	0.2584

TABLE XVI: Pure decoding efficiency

Threads File sizes	1	2	4	6	8	10	12	16	20	24
1 MiB	1.0	0.9006	0.5154	0.3487	0.3568	0.2845	0.2554	0.1758	0.1518	0.1379
5 MiB	1.0	0.3290	0.1175	0.1574	0.1127	0.2416	0.1109	0.1446	0.1049	0.2083
10 MiB	1.0	0.8967	0.7645	0.6835	0.5286	0.4492	0.4369	0.3764	0.3766	0.3507
50 MiB	1.0	0.9553	0.8538	0.7928	0.6047	0.5385	0.4745	0.4543	0.4234	0.4181
100 MiB	1.0	0.1030	0.1155	0.1062	0.0694	0.0694	0.0686	0.1492	0.0681	0.0775
500 MiB	1.0	0.9719	0.8855	0.8230	0.6524	0.5612	0.5255	0.4747	0.4627	0.4504
1 GiB	1.0	0.9765	0.8963	0.8287	0.7771	0.5682	0.5317	0.4856	0.4711	0.4639
5 GiB	1.0	0.3712	0.3394	0.2959	0.3121	0.2802	0.2635	0.2393	0.2346	0.2331
10 GiB	1.0	0.9608	0.9281	0.6509	0.2109	0.5620	0.5228	0.4833	0.4665	0.4601

E. Cluster testing

TABLE XVII: Overall encoding times for different folders on cluster

	Processes	1			2			4		
Source	Threads	1	2	4	1	2	4	1	2	4
Numpy										
	Encoding times	273.1562	207.1372	222.3222	160.9471	116.7891	124.3936	160.9471	116.7891	124.3936
	Encoding efficiency	1.0000	0.6594	0.3072	0.8486	0.5847	0.2745	0.7875	0.2330	0.1450
	Decoding times	97.3082	76.6709	98.8446	57.8702	62.0716	72.3069	52.7536	54.1637	65.1009
	Decoding efficiency	1.0000	0.6346	0.2461	0.8407	0.3919	0.1682	0.4611	0.2246	0.0934
Pytorch										
	Encoding times	1329.5976	2536.5453	1737.9396	524.3115	780.8548	522.8379	483.0258	458.4398	419.2587
	Encoding efficiency	1.0000	0.2621	0.1913	1.2679	0.4257	0.3179	0.6882	0.3625	0.1982
	Decoding times	953.3874	813.2746	782.2937	203.3535	460.1837	495.1923	286.6861	275.3211	310.8803
	Decoding efficiency	1.0000	0.5861	0.3047	2.3442	0.5179	0.2407	0.8314	0.4329	0.1917

F. Local testing

TABLE XVIII: Overall encoding times for different folders on local machine

	Processes	1	2	4
Source	Threads	1	1	1
Numpy				
	Encoding times	2.0731	1.0857	0.621
	Encoding efficiency	1.0	0.9547	0.8345
	Decoding times	1.8602	1.0178	0.5757
	Decoding efficiency	1.0	0.9138	0.8078
Pytorch				
	Encoding times	15.1102	7.7887	5.4025
	Encoding efficiency	1.0	0.97	0.6992
	Decoding times	14.6897	7.7248	5.6166
	Decoding efficiency	1.0	0.9508	0.6538
Linux				
	Encoding times	85.0387	49.5736	30.6647
	Encoding efficiency	1.0	0.8577	0.6933
	Decoding times	80.3765	47.8856	29.6687
	Decoding efficiency	1.0	0.8393	0.6773