Parallelized Huffman Coding for Image Compression

Martin Ma[1], Siyao Li[1], Gekai Liao[1], and Victor Zhu[1]

[1]Harvard University

May 4, 2023

Abstract

Huffman coding is a well-known method for lossless data compression that has various applications. Our project aims to enhance the performance of image compression by parallelizing the current C++ implementation of Huffman coding. To achieve this, we have utilized OpenMP shared memory parallelism to optimize the frequency analysis stage of Huffman coding, and attempted to parallelize the file writing process using MPI I/O and HDF5. Our efforts have led to a successful reduction of processing time by 9.5% while employing 24 threads. Furthermore, we have presented adaptive parallelization, a novel approach to dynamically assess whether an image is worth the compression computing resource.

1 Background and Significance

Huffman encoding is a lossless data compression algorithm that uses optimal prefix code. It was developed by David Huffman in his paper "A Method for the Construction of Minimum-Redundancy Codes"[2]. It generates a variable-length code table for encoding source symbols based on the estimated probability or frequency of occurrence for each possible value of the symbol. This results in more common symbols being represented using fewer bits than less common symbols. The algorithm can be implemented efficiently and finds a code in time linear to the number of input weights if these weights are sorted[3].

In the project, we implemented a huffman encoding for images. It allows for efficient storage and transmission of images while preserving their quality. Images often contain large amounts of data, and storing or transmitting them can be time-consuming and expensive. By using Huffman encoding, the amount of data required to represent the image can be significantly reduced, resulting in a smaller file size and faster transmission times.

This type of compression is frequently employed in the field of medicine, especially as part of the DICOM standard which is backed by major medical equipment manufacturers, for applications such as ultrasound machines, nuclear resonance imaging machines, MRI machines, and electron microscopes. Variants of the Lossless algorithm are also commonly used in the RAW format, which is popular among photography enthusiasts due to its ability to retain information from a camera's image sensor without any loss of data.

In Figure 1, we gave an example of how to construct a huffman tree given the occurrences of each letter. In this hypothetical example, letter A occurs 15 times; letter B occurs 7 times; letter C and D occurs 6 times; letter E occurs 5 times. In every iteration, we form a tree node by combining the two letters that have the lowest number of occurrences. In this example, D and E have the lowest number of occurrences. Thus, in the first iteration, we combine D and E to a new node that have 11 occurrences in total. In the next rotation, we combine B and C. Then, we combine the node BC with DE to form

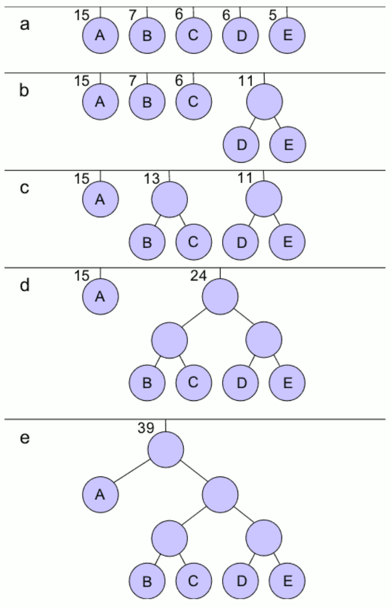

Figure 1: Example of constructing a huffman tree

the tree. If we continue with the process, we will have a huffman tree fully encoded. If the left branch is thought as "0" and right branch as "1", then A = "0", B = "100", C = "101", D="110", E = "111".

Originally, we need at least 3 bits to represent a letter because there are five distinct letter. The number of bits we originally need is $3 * (15 + 7 + 6 + 6 + 5) = 117$. Now, with huffman encoding, we only need $15 + 3 * (7 + 6 + 6 + 5) = 87$ bits, lowering the storage size by 25 percent.

The advantage of parallelized Huffman encoding is that it can significantly reduce the time required to encode large datasets. Traditional Huffman encoding operates on a single thread, which can become a bottleneck when dealing with large amounts of data. Parallelized Huffman encoding, on the other hand, distributes the work across multiple threads or processors, allowing for faster encoding times. This implies we could do meaningful work on video compression where videos are comprised of numerous image frames. In this way, compressing video can be achieved through huffman encoding.

2 Scientific Goals and Objectives

The primary scientific goal of parallelized Huffman encoding is to optimize the performance of the encoding process. This involves designing and implementing efficient parallel algorithms that can take advantage of modern computing architectures, such as multi-core CPUs, GPUs, and clusters. In this context, we aim to improve the performance by using a combination of OpenMP and MPI. By improving the performance of Huffman encoding, researchers can enable faster and more efficient data processing in a variety of applications, such as data compression, image processing, and scientific computing.

Another important scientific objective for parallelized Huffman encoding is to achieve scalability. Scalability refers to the ability of an algorithm to maintain its performance as the size of the dataset increases. In the context of Huffman encoding, this means developing parallel algorithms that can efficiently encode very large datasets without losing performance or efficiency. Achieving scalability is critical for many scientific applications, such as large-scale data analysis, genomics, and climate modeling. This is also important if we want to have a good performance for video compression. Using a HPC architecture will help us achieve this objective as it is capable of processing massive amounts of data in a short amount of time, which makes it ideal for applications that require the processing of large datasets.

3 Algorithms and Code Parallelization

In this section, we will outline the algorithms used for Huffman coding algorithm, and introduce a technique, called adaptive parallelization, to conserve computational resources.

Huffman Coding Algorithm

The sequential version of Huffman coding is adopted from the GitHub[4]. We chose this code repository because it was one of few C++ implementation of Huffman coding found on GitHub. We added parallel features and improved the performance of Huffman coding process. The code is composed of the following two parts:

- **Compressor.cpp**: The file compression is handled by the **Compressor.cpp** code, which is divided into two parts. The first part calculates the encoding information and comprises the following five sections:

  1. Retrieving the size information of the original file, and writing some metadata to the compressed file.
  2. Conducting the frequency analysis by counting the frequency of each image pixel in the original image and generating an array of size 256 that contains the frequency information.
  3. Creating an empty array to hold the Huffman tree.
  4. Creating a Huffman Tree for every unique pixel value.
  5. Repeatedly combining the two nodes with the smallest weights until there is only one node left. Each leaf node in the tree represents a character in the text or data stream, and the frequency of that character determines the weight of the node.

  The subsequent step in the compression process entails creating a compressed file with the encoding match table obtained from the previous stage. The original file is read once again, pixel by pixel, and the corresponding encoding value for each pixel is looked up. The encoded value is then written to the compressed file.

- **Decompressor.cpp**: This code is responsible of decompressing the file generated by **Compressor.cpp** and it is basically the reverse of compressing a file.

Frequency Analysis Parallelization

The frequency analysis part of the algorithm has little to no data dependency for each pixel step, making it highly parallelizable. This feature serves as a significant advantage for achieving high parallel efficiency and speedup when implementing the parallelized version of the Huffman encoding algorithm.

In our implementation, we utilized OpenMP shared memory parallelism by dividing the original image file into chunks that can be accessed by different threads. Each thread is assigned a corresponding starting and ending index to access the specific chunk and saves the frequency of the sub-portion in a private array. At the end of the parallelized section, an atomic reduction operation is performed among all the threads to output the final frequency array.

Parallel File Writing

Similar to the frequency analysis part of the algorithm, the file writing section also has minimal data dependency since it involves simply looking up the encoded value for each pixel and writing it sequentially to the compressed file. This step presents ample potential for enhancing efficiency by reducing the I/O overhead of the Huffman coding algorithm.

We made two attempts to improve the parallel writing performance of the algorithm by using MPI I/O and MPI with HDF5 implementation. However, in both cases, the parallel writing performance was worse than sequential writing. We investigated the possible sources of error and concluded that the communication overhead might be the bottleneck in this case. These findings emphasize the importance of carefully considering the trade-offs between computation and communication overheads when designing parallel algorithms.

Adaptive Parallelization

In this section, we discuss the efficiency of Huffman coding in compressing images and why, in some instances, the compressed image size is larger than the original image. We also propose a method to determine when to use compression based on the estimated compression ratio.

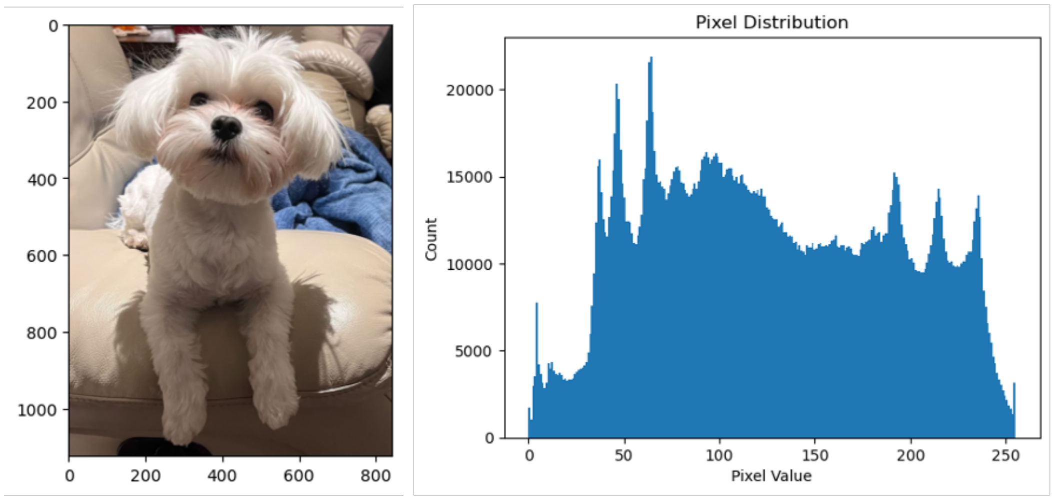

Figure 2: Pixel distribution of an example image, with calculated entropy of 7.9.

The efficiency of Huffman coding depends on the pixel distribution of the original image. In an uncompressed image, each pixel value is represented by an 8-bit number. After compression, the average number of bits per value can be approximated by the entropy of the pixel distribution, as shown in Equation 1. The efficiency of the compression is dictated by the entropy value, with a lower entropy resulting in a more efficient compression. The example image in Figure 2 has an entropy of 7.9.

$$H = -\sum_{i=1}^{N} p(i) \log_2 p(i) \tag{1}$$

To illustrate the impact of entropy on compression efficiency and understand the range of entropy, two extreme cases of image pixel distribution are considered. The left subfigures of Figure 3 corresponds to a randomly generated image. In this scenario, the pixel distribution is uniform, resulting in an entropy of 8.0. This means no compression is achieved, and due to the memory required to store the Huffman tree, the compressed file size may be larger than the original image. On the other hand, the right subfigures of Figure 3 corresponds to a highly structured image. In this case, the image is divided into two halves, with the left half being all white and the right half being all black. The pixel distribution is highly bimodal, requiring only 1 bit to represent each pixel. This results in an ideal compression ratio of 8 times.

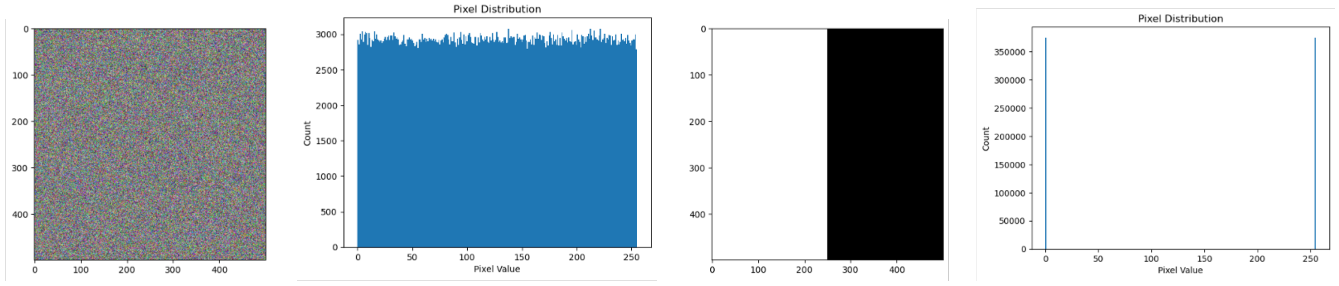

Figure 3: The left 2 subplots are a randomly generated image and its corresponding uniform pixel distribution, with entropy = 8.0. The right 2 subplots are a black & white image and its corresponding Bernoulli pixel distribution, with entropy = 1.0.

To optimize the use of Huffman coding for image compression, we propose an adaptive compression strategy based on the estimated compression ratio, shown in Equation 2. For each image, the entropy can be calculated to estimate the compression ratio, which is determined by dividing the original size by the entropy. If the compression ratio is greater than 1.5, the image should be compressed, as the reduction in file size will be significant. If the compression ratio is less than or equal to 1.5, the image should not be compressed to save computational time and resources. This method helps in conserving computational resources and avoiding situations where the compressed image size is larger than the original image.

$$C = \frac{8.0}{H} \tag{2}$$

Validation, Verification

As Huffman Coding is a lossless compression algorithm, validating and verifying the parallel Huffman encoding algorithm for image compression requires ensuring that the compressed and decompressed images accurately reflect the original image in terms of visual quality and information content. We tested the algorithm's output against the expected results for a range of different images and input parameters. Specifically, we ran the algorithm on a set of test images with varying levels of detail or size, and compared the resulting compressed images with the original images at a pixel level. Overall, the validation and verification process demonstrated the accuracy and robustness of our parallel Huffman encoding algorithm for image compression.

4 Performance Benchmarks and Scaling Analysis

To evaluate the performance of the parallelized Huffman algorithm for image compression proposed in Section 3, we conducted a series of experiments using a large raw image with a file size of 36.2 MB. This image size is representative of those typically generated by professional photographers, and thus, serves as a suitable benchmark for assessing our algorithm's effectiveness in real-world scenarios.

The sequential baseline for processing the image took 6.2 seconds, with the time breakdown for various components of the process illustrated in Figure 4. Frequency analysis accounted for 12% of the total processing time, while image encoding and writing together contributed to 88% of the time. In contrast, the construction of the Huffman tree only added a negligible amount to the total run time due to the limited number of possible values for image pixels, which is capped at 256. As the size of the image increases, the Huffman tree construction can be considered a constant-time operation, since its computational complexity remains relatively unchanged.

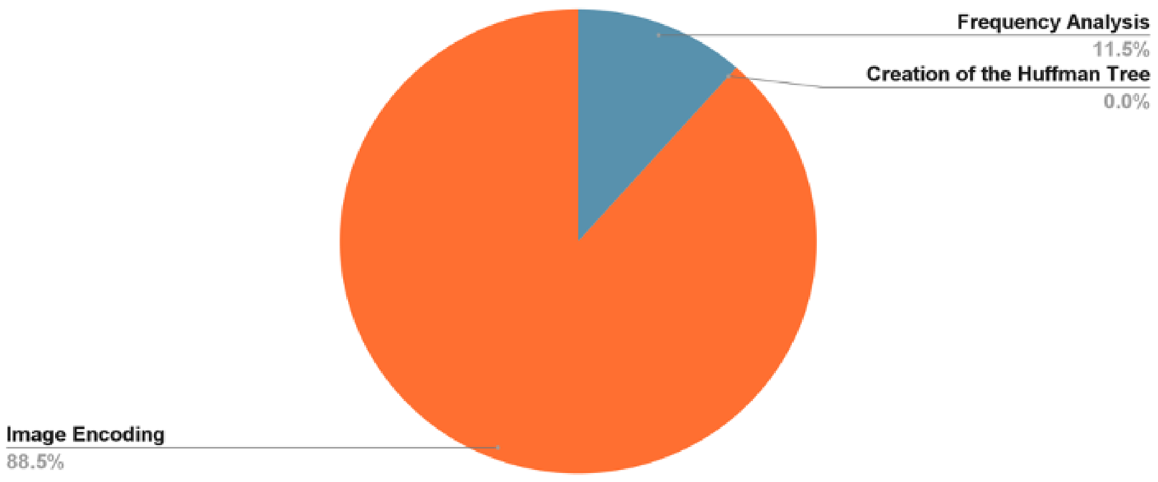

Figure 4: Sequential code time breakdown to compress a 36.2 MB raw image.

Although our project is not concerned with floating point operation, we include the roofline model for our hardware in Figure 5 for the completeness of the project. Since, most of the operation is reading and counting, number of FLOP doesn't corresponds to performance. Another metrics such as strong scaling is better suited as the evaluation metric.

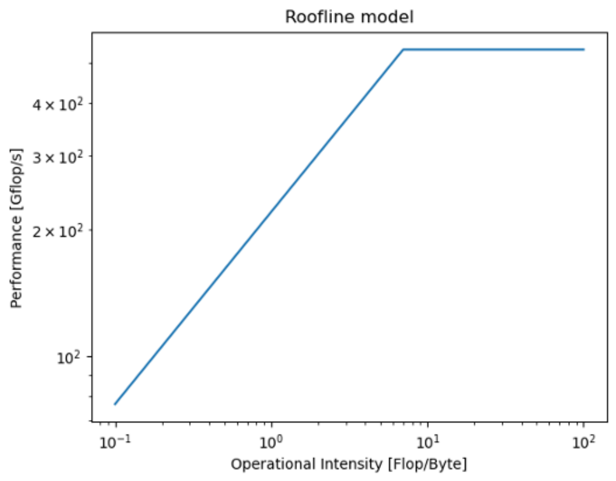

Figure 5: Roofline model for Intel Xeon Processor E5-2683 v4

Figure 4 highlights the primary bottlenecks in the sequential baseline as image encoding, writing, and frequency analysis. To address these limitations, our proposed algorithm specifically targets these areas for improvement by introducing parallelization techniques. We conducted an ablation study to demonstrate the effectiveness of parallelizing different components of the algorithm, with the results presented in Table 1.

	Time [s]
Sequential baseline	6.2 (100%)
+ parallel frequency analysis	5.2 (84%)
+ parallel writing	5.2 (84%)

Table 1: Ablation study using 4 threads and 2 processes

Theoretically, the most significant improvement should be observed in the parallel writing component, which can be attributed to the inherent benefits of parallel processing. By dividing the writing tasks into smaller segments and concurrently processing them across multiple cores or processing units, the overall time spent on file I/O operations should be substantially reduced. This results in a more efficient use of system resources and enables the algorithm to complete the writing tasks in a shorter time frame, effectively addressing the primary bottlenecks identified in the sequential baseline. However, when we tried to use both MPI with HDF5 and MPI-IO for parallel writing, the performance didn't increase. This is potentially due to the communication overhead as the multi-process regime doesn't allow the shared memory, and the file to be written needs to be sent over the network. Another cause of no improvement of performance may be the file system caching behavior. The file system may cache writes and buffer them before committing them to disk, which can make the actual write operations slower than expected. In practice, we observed the most significant performance improvement in the multi-thread frequency analysis section, which resulted in a 6% reduction in runtime when using 4 threads.

To further assess the performance of our parallelized Huffman algorithm for image compression, we performed a strong scaling analysis. Strong scaling analysis is a method of evaluating the algorithm's speedup and efficiency as the number of threads or processing units increases, while keeping the problem size constant [5]. The goal of strong scaling is to achieve a linear speedup, meaning that doubling the number of threads should ideally result in halving the computation time. Another metrics to measure the effectiveness of the parallel algorithm is efficiency, an ideal efficiency value is 1, indicating that the parallel algorithm is effectively utilizing all the available threads.

We conducted experiments using the same 36.2 MB raw image and executed the parallelized algorithm on a range of thread configurations, from single-threaded execution to utilizing multiple threads. The speedup and efficiency for each configuration are calculated and analyzed to determine the scalability of our algorithm.

The strong scaling analysis results is plotted in Figure 6 and analyzed to identify trends in speedup and efficiency as the number of threads increased. This allows us to observe whether our parallelized Huffman algorithm exhibits good scalability and can effectively utilize additional processing resources. From the analysis, we can draw conclusions about the optimal thread configuration for our algorithm and identify any potential limitations in its scalability.

After analyzing the speedup and efficiency graph presented in Figure 6, it can be observed that the parallelized Huffman algorithm achieves higher speedup as the number of threads increases. However, the obtained speedup falls short of the ideal performance we would hope to achieve, given the resources utilized. The observed results are not surprising since the baseline benchmark data exhibits a considerable amount of writing runtime, and the improvement in parallel writing is constrained in our parallelized algorithm. In terms of efficiency, a steep drop from 1 to 2 threads is observed, followed by relatively consistent efficiency across subsequent threads. However, the observed efficiency is not ideal, primarily because only a small part of the algorithm, i.e., the frequency analysis portion, can be effectively parallelized, and it accounts for a small portion of the total runtime. The strong scaling analysis provides a comprehensive assessment of the parallelized Huffman algorithm's performance and scalability, showcasing its potential for efficient image compression in real-world applications.

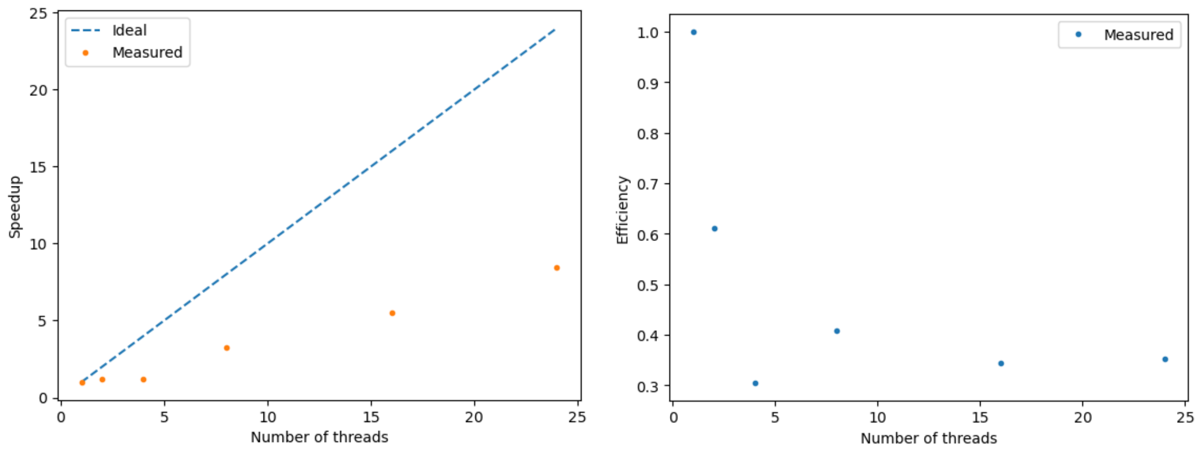

Figure 6: Strong scaling analysis

We also conducted a benchmark for the parallelized file-writing section of the algorithm, but surprisingly, it did not result in any significant improvement in runtime. We set the number of threads to 1 and increased the number of processes involved in the compression file-writing stage. The runtime slowed down as we increased the number of processes available for parallel writing. This can be attributed to the increase in communication calls among processes needed to transfer data, resulting in extra communication overhead. In our specific case, no computation takes place in between the writing process, and as a result, no overhead was hidden. Hence parallel writing did not contribute to the speed-up of the parallel Huffman encoding algorithm.

Moreover, the adaptive parallelization technique enables further optimization by estimating the compression ratio and only compressing when the ratio exceeds a certain threshold. To assess this, we compare the parallel algorithm both with and without adaptive parallelization on a folder containing 30 images. The integration of adaptive parallelization reduces computation time by 20% while only increasing the final compressed image size by a mere 2%.

5 Resource Justification

The core hours required to run and evaluate our code determine the amount of resources needed. This can be calculated by multiplying the number of cores by the wall time of a production run, which gives the value of node hours. This highlights the importance of using a high-performance computer architecture. The compute cluster we use has 1 node and 32 cores running on a Broadwell Architecture, which can be found at Figure 7.

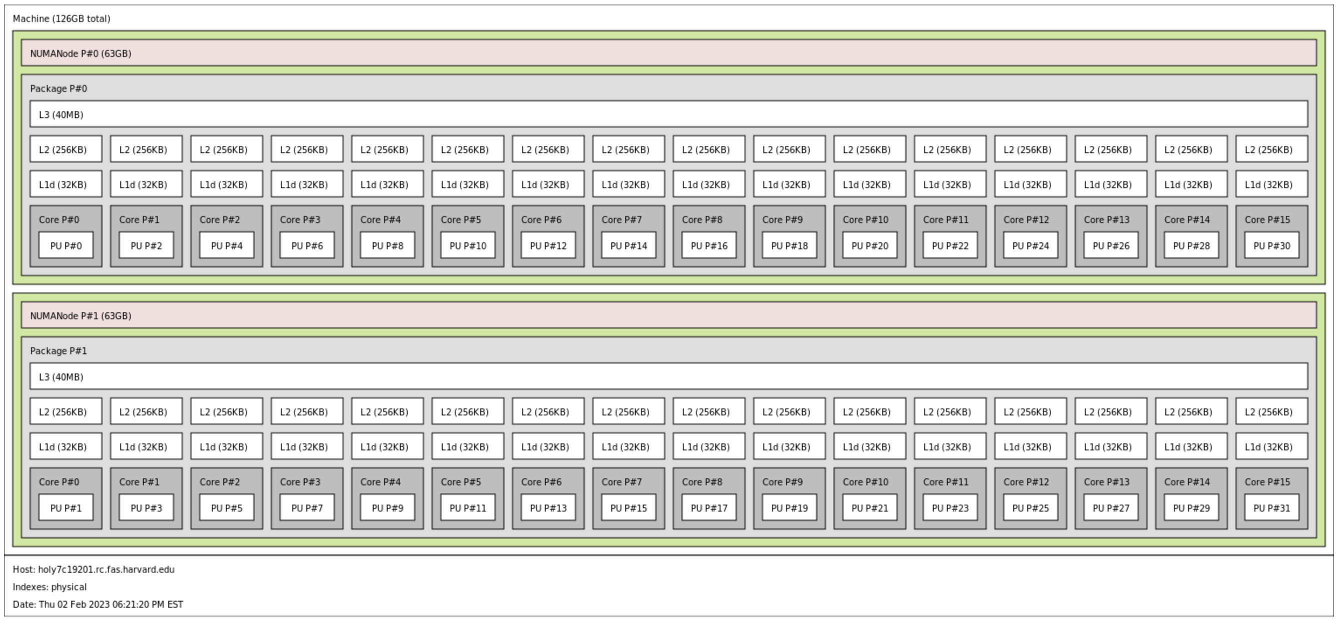
Machine (126GB total)
NUMANode P#0 (63GB)
Package P#0
L3 (40MB)
L2 (256KB)
L1d (32KB)
Core P#0 Core P#1 Core P#2 Core P#3 Core P#4 Core P#5 Core P#6 Core P#7 Core P#8 Core P#9 Core P#10 Core P#11 Core P#12 Core P#13 Core P#14 Core P#15
PU P#0 PU P#2 PU P#4 PU P#6 PU P#8 PU P#10 PU P#12 PU P#14 PU P#16 PU P#18 PU P#20 PU P#22 PU P#24 PU P#26 PU P#28 PU P#30
NUMANode P#1 (63GB)
Package P#1
L3 (40MB)
L2 (256KB)
L1d (32KB)
Core P#0 Core P#1 Core P#2 Core P#3 Core P#4 Core P#5 Core P#6 Core P#7 Core P#8 Core P#9 Core P#10 Core P#11 Core P#12 Core P#13 Core P#14 Core P#15
PU P#1 PU P#3 PU P#5 PU P#7 PU P#9 PU P#11 PU P#13 PU P#15 PU P#17 PU P#19 PU P#21 PU P#23 PU P#25 PU P#27 PU P#29 PU P#31
Host: holy7c19201.rc.fas.harvard.edu
Indexes: physical
Date: Thu 02 Feb 2023 06:21:20 PM EST

Figure 7: Architecture of the Broadwell Node

The amount of computational resource required is dependent on the original file size. For example, if we have a 5-minute 4k videos with 24 frame per second. With our calculations in Section 4, the amount of time takes to compress this video frame-by-frame will be 12 hours. Using more threads would certainly improve the performance. Assuming a speedup of 10x, the number of core hours can be calculated below:

$$38.4 \text{ node hours} = 32 \text{ cores} \times 1 \text{ node} \times \frac{4320s}{3600\frac{s}{\text{hour}}}$$

Our benchmarks consisted of running compression on videos with different length under sequential and parallelized code. Table 2 shows the required core hours per run.

Video Length (minute)	Sequential Baseline (core hours)	Parallelized (core hours)
1	2.4	0.26
2	4.8	0.53
3	7.2	0.80
4	9.6	1.07
5	12	1.33
6	14.4	1.60
7	16.8	1.87
8	19.2	2.13
9	21.6	2.4
10	24	2.7

Table 2: Justification of the resource request

The values can be summed to get total number of compute hours for the experiments while considering the 32 cores in the parallelized time. The experiment above will request a total of 601 core hours to complete in the supercomputing center.

References

[1] Leslie Lamport. *LaTeX: a document preparation system.* Addison-Wesley, Reading, Massachusetts, 1993.

[2] Huffman, D. *Huffman, D. (1952). "A Method for the Construction of Minimum-Redundancy Codes".* Proceedings of the IRE. 40 (9): 1098–1101. doi:10.1109/JRPROC.1952.273898.

[3] Van Leeuwen, Jan *(1976). "On the construction of Huffman trees" (PDF). ICALP: 382–410.* Retrieved 2014-02-20.

[4] Ersel Hengirmen, *C++ Implementation of Huffman Coding*, (2021), GitHub Repository, https://github.com/e-hengirmen/Huffman-Coding

[5] M. D. Hill and M. R. Marty, *"Amdahl's Law in the Multicore Era," in Computer, vol. 41, no. 7, pp. 33-38, July 2008, doi: 10.1109/MC.2008.209.*