CS5285 – Big Data Algorithms and Technologies

Parallel Color Quantization

A *Big Data* Approach to Image Segmentation

Based on ***Joblib***, ***Hadoop*** and ***Spark***

Final Report

by Group 11

Advisor: Prof WONG Ka Chun

City University of Hong Kong
Department of Computer Science

Table of Contents

Abstract **1**

1 Introduction **1**

1.1 Background of Color Quantization 1

1.2 Motivation and Objective of this Project 2

1.3 Overview of Dataset and Our Methodology 3

1.4 Hardware and Software Environment in this Project 4

2 Baseline k-Means Implementation in Python **5**

2.1 Theory of the k-Means Algorithm 5

2.2 The Implementation in Our Project 7

2.3 Time Cost for Baseline Python Experiment 8

3 Parallel Refactoring of Baseline Experiment using Joblib **9**

3.1 Parallel Alignment of Nearest Centroids 9

3.2 Parallel Updating of Centroids 10

3.3 Time Cost for Parallel Python Experiment 10

4 Parallel Implementation Based on Hadoop and Spark **12**

4.1 Calculate the Centroids based on Hadoop 12

4.2 Assign cluster centers to each pixel Based on Spark 15

4.3 Time Cost for Hadoop with Spark Experiment 16

5 Parallel Implementation Based on Pure Spark **17**

5.1 Methodology of Parallel Computing using Spark 17

5.2 Time Cost for Pure Spark Experiment 18

6 Results and Conclusions **20**

6.1 Time Cost Comparisons 20

6.2 Effect of Color Quantization on Image Properties 23

6.3 Conclusion 25

7 Further Discussion **26**

7.1 Discussion on Our Results and Conclusions 26

7.2 Potential Directions for Future Research 26

8 References **27**

9 Individual Contribution **28**

10 Dataset, Code and Result Availability **29**

Abstract

Color quantization is a method of image segmentation that represents an image using a limited number of colors. Given that an image may contain millions or even billions of pixels, traditional sequential processing algorithms can result in significant time overhead. In this study, we leverage big data platforms such as Hadoop and Spark to implement image color quantization based on k-means clustering through parallel computing. By comparing the results of traditional methods and parallel computing, our experimental results demonstrate that our new method significantly reduces time overhead. Additionally, we compared the effects of varying parameters in the color quantization algorithm, such as the number of colors, on image quality and runtime. We also provide a discussion related to the practical application of color quantization.

Keywords: color quantization, parallel computing, k-means, Hadoop, Scala

1 Introduction

1.1 Background of Color Quantization

Image segmentation, a crucial process in computer vision, involves partitioning an image into multiple segments to simplify its representation and make it more meaningful for analysis. Color quantization is a method of image segmentation that allows a picture to be represented by only a number of colors while preserving the basic structure of the image[1]. Realistic images often contain complex variations in color[2], which can make economical descriptions difficult, yet human observers can readily reduce the number of colors in images to a small proportion they judge as relevant[3]. Here, we take an artificially generated image as an example.

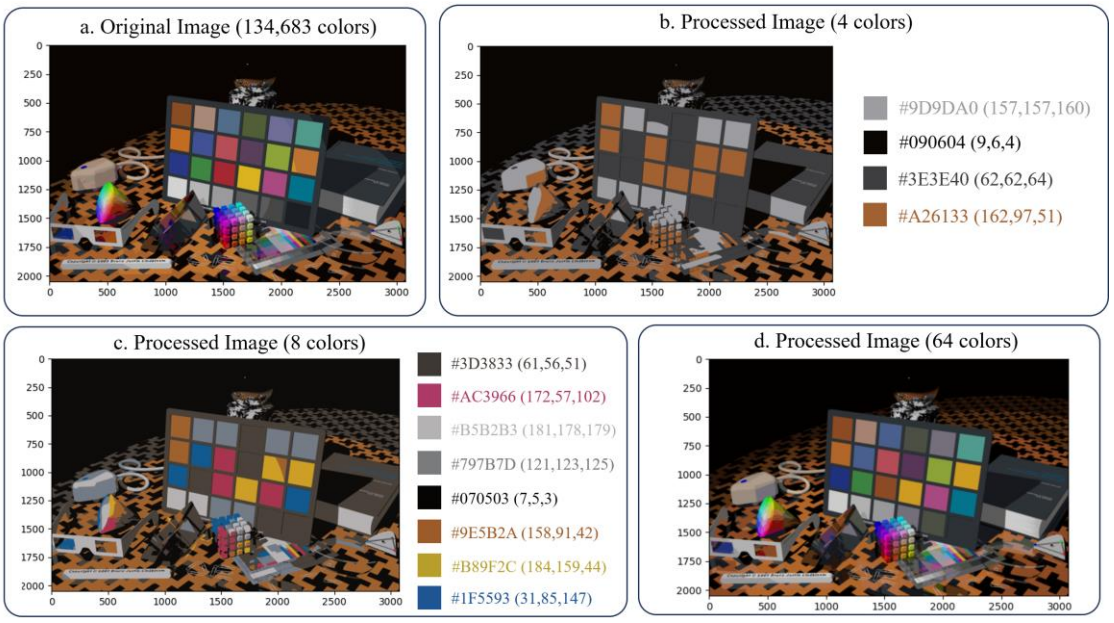

Fig 1.1 - Representing the same picture with different numbers of colors

For the original image shown in Figure 1.2.a, it comprises a total of 6,291,456 distinct pixels with 134,683 unique RGB colors. In the context of a machine learning analysis pertaining to image color, utilizing all 134,683 colors as features would result in substantial memory and time overhead during both training and prediction phases. This could potentially lead to catastrophic resource wastage.

Color quantization is a possible solution to this issue. As demonstrated in Figure 1.1.b, the image is effectively segmented using merely four colors, while preserving the basic structure of the image. In Figure 1.1.c, eight different colors are employed to represent the image, thereby enhancing the level of detail and richness of color. Figure 1.1.d uses 64 different colors to represent the image. Despite the image appearing nearly identical to the original, the number of colors has been reduced by more than 2100 times! This approach significantly optimizes resource usage without compromising the quality of the image representation.

1.2 Motivation and Objective of this Project

While existing color quantization algorithms yield satisfactory results, a study conducted in April 2023 pointed out a significant unresolved issue about running time in the field of color quantization[2]. This issue arises from the advancements in photographic technology, which have led to the production of images of increasingly superior quality. The problem lies in the substantial time consumption required for color quantizing high-resolution images, which poses a considerable challenge in the field.

A typical photograph may encompass millions to billions of pixels, and processing such an image entails managing these vast quantities of pixels. This substantial number, when processed sequentially, incurs significant time overheads. Handling such extensive data for segmentation and color quantization is both computationally intensive and time-consuming, particularly for tasks like pixel mapping and cluster allocation. Thus, striking a balance between the precision of color representation and the efficiency of the quantization process poses a formidable challenge.

The integration of parallel computing into image processing, specifically for color quantization in image segmentation, presents a contemporary solution to the challenges posed by big data. The computational demands associated with processing large-scale images necessitate the use of parallel computing solutions such as Joblib, Hadoop, and Spark for efficient data management.

Therefore, we aim to develop a parallel image color quantization technique tailored for high-resolution images. This method will empower computers to process image color quantization efficiently, leading to the computation of diverse color feature representations and the generation of new image variants. This project seeks to integrate these advancements with the k-means clustering algorithm, thereby proposing an innovative solution for image segmentation that caters to the requirements of big data.

1.3 Overview of Dataset and Our Methodology

For an RGB image with n bits, it may have n^3 different colors. Getting such a large number down to a small range is a challenge. Some studies have proposed the use of clustering as a means of color quantification[1,2], and is considered to have good effects. The k-Means algorithm, a well-known implementation of a clustering algorithm, can be adapted for parallel processing. Existing research has substantiated the efficacy and speed of parallel versions of the k-Means algorithm[4]. This highlights the potential for leveraging parallel computing to enhance the performance of such algorithms.

Our work is structured around a sequence of experiments, each building upon the last, to refine our approach and ensure efficient processing of image data, which may range from millions to billions of pixels. The figure above shows the workflow of our experiments.

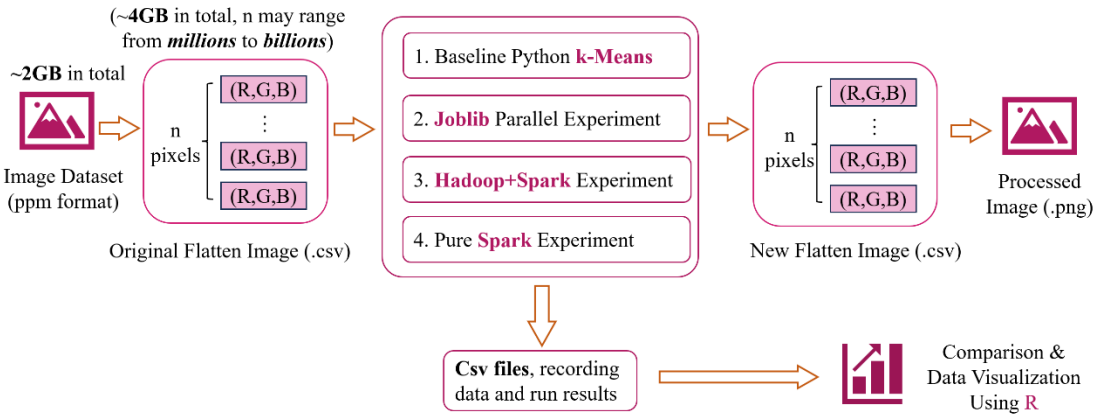

Fig 1.2 - Workflow for our color quantization experiment using big data technologies

We used the image dataset from the Image Compression Benchmark website[5], which comprises high-resolution, high-precision photographic images in Portable Pixmap Format. We selected a collection of color RGB images among them, which have 16-bit or 8-bit original file color bits. The raw dataset is approximately 2GB in ppm format. We decompose these images into RGB vectors and store them as .csv files for analysis, with each row representing pixel data by its RGB components. The image data in .csv files exceed 4GB in size.

We have structured our experimental approach into four distinct phases: Baseline Python k-Means, Joblib Parallel Experiment, Hadoop and Spark Experiment and Pure Spark Experiment. We establish a baseline by applying a Python implementation of the k-Means algorithm. By utilizing the Joblib library, we embark on our first parallel computing experiment. To expand our scope, we integrate Hadoop's robust data handling capabilities with Spark's in-memory processing to manage larger datasets more effectively. We have implemented MapReduce framework using Hadoop, and the

Resilient Distributed Dataset implementation is also utilized using Spark. Finally, we use pure Spark to focus on optimizing in-memory data processing.

For programming language, we coded program Baseline Experiment and Parallel Python Experiment using Python 3.8. For Hadoop and Spark parts, we used Java 11 and Scala 3.3 respectively.

Each experiment generates new CSV files, including files recording color quantization data and execution details, and processed flatten image data. We used a unified script to convert image data from csv format and image format to each other.

For further analysis, we use R to visualize and scrutinize the data. In this way, we aim to demonstrate the effectiveness of parallel computing in the field of image color quantization.

1.4 Hardware and Software Environment in this Project

Initially, we planned to use a virtual machine or cloud server for our experiment. However, due to potential performance issues, we opted for a personal laptop running Windows 11 Professional. The laptop has an Intel i7-12700H processor with 14 cores and 20 threads, a 2TB SSD, and 32GB DDR5 memory. Although the laptop has an RTX-3070ti GPU and CUDA 11.2 environment, we conducted the experiment solely on the CPU for broader compatibility.

The experiment requires Python 3.8 or higher and Java Runtime Environment 11. For baseline and parallel Python experiments, we suggest using a virtual environment. For Hadoop and Spark experiments, Hadoop 2.7 and Spark 3.2.4 are needed. Ensure JAVA_HOME, HADOOP_HOME, and SPARK_HOME paths are set in the environment variables and start all Hadoop services with the start-all command before running the project. If you need to compile your program, install Java Development Kit 11 and Scala 3.3.1, and ensure Maven is available for dependency management in your project.

2 Baseline k-Means Implementation in Python

Here we have designed a baseline experiment using the k-means algorithm in python.

2.1 Theory of the k-Means Algorithm

The goal of k-means clustering is to divide n points into k clusters such that each point is closest to the center of the cluster (centroid). Corresponding to an image file, we can consider each pixel as a point and the RGB value of each pixel as its 3D coordinates. Thus, each cluster center can then be considered as a quantized color of the image.

The implementation of k-means starts with initializing a set of random centroids. The pseudo-code for initializing k centroids is as follows:

Algorithm 2-1 Generate k Random Initial Centroids

Input: k

Output: the initial k centroids

```
1   Function GenerateRandomDistance(k)
2       centroids = EmptyList
3       For i=0 To k-1 Do
4           r = RandomInteger(0, 255)
5           g = RandomInteger(0, 255)
6           b = RandomInteger(0, 255)
7           newCentroid = [r,g,b]
8           centroids.append(newCentroid)
9       End For
10      Return centroids
11  End Function
```

We need to calculate the distance between points and points for knowing the distance between points and centroids as a way to categorize points into different clusters. The formula for calculating the Euclidean distance between two points A(X_a, Y_a, Z_a) and B (X_b, Y_b, Z_b) is as follows:

$$D_{a,b} = \sqrt{(X_a - X_b)^2 + (Y_a - Y_b)^2 + (Z_a - Z_b)^2}$$

The pseudo-code is given below:

Algorithm 2-2 Calculate the Euclidean distance between two pixels

Input: pixel_1, pixel_2

Output: the Euclidean distance between pixel_1, pixel_2

```
1    Function CalculateDistance(pixel _1, pixel _2)
2        dimension = Size of pixel _1 // dimension = 3
3        distance = 0
4        For i = 0 To dimension-1 Do
5            distance += (pixel _1[i] –pixel _2[i])^2
6        End For
7        Return SquareRoot(Distance)
8    End Function
```

After calculating the distances from all points to centroids, we record for each point the centroid closest to them, categorizing them. Then for each category, the average of all the coordinates in pixels in this category is calculated and they are used as new centroids: The formula for calculating the average value of the coordinates is as follows:

$$(\bar{x}, \bar{y}, \bar{z}) = (\frac{1}{n}\sum_{i=1}^{n} x_i, \frac{1}{n}\sum_{i=1}^{n} y_i, \frac{1}{n}\sum_{i=1}^{n} z_i)$$

Considering the iterative process of k-means, we set termination conditions for it. We keep the centroids of each round of computation and determine whether the centroids of the latest round converge with the previous round, i.e., whether the Euclidean distance between each pair of centroids is less than a threshold (default 5). If the centroids converge, the program terminates the k-means iterations. The pseudocode is shown below:

Algorithm 2-3 Determine whether the new centroids have converged

Input: oldCentroids, newCentroids

Output: **True** if the centroids are converged, **False** if not

```
1    Function isConverged(oldCentroids, newCentroids)
2        k = oldCentroids.length
3        For i = 0 To k Do
4            For j = 0 To k Do
5                If CalculateDistance(oldCentroids[i], newCentroids[j]) > 5 Then
6                    Return False
7                End If
8            End For
9        End For
10       Return True
11   End Function
```

2.2 The Implementation in Our Project

In our project, there are 2 steps in k-means process – assigning pixels to the nearest cluster center and calculating the average of each set of pixels as the new cluster center. Since we use RGB values instead of coordinates, so for the formula of the Euclidean distance between two pixels A(X_a, Y_a, Z_a) and B (X_b, Y_b, Z_b) is as follows:

$$D_{a,b} = \sqrt{(R_a - R_b)^2 + (G_a - G_b)^2 + (B_a - B_b)^2}$$

The formula for calculating the average RGB value of a group of pixels is as follows:

$$(\bar{R}, \bar{G}, \bar{B}) = (\frac{1}{n}\sum_{i=1}^{n} R_i, \frac{1}{n}\sum_{i=1}^{n} G_i, \frac{1}{n}\sum_{i=1}^{n} B_i)$$

The flowchart of computing the cluster center is shown below:

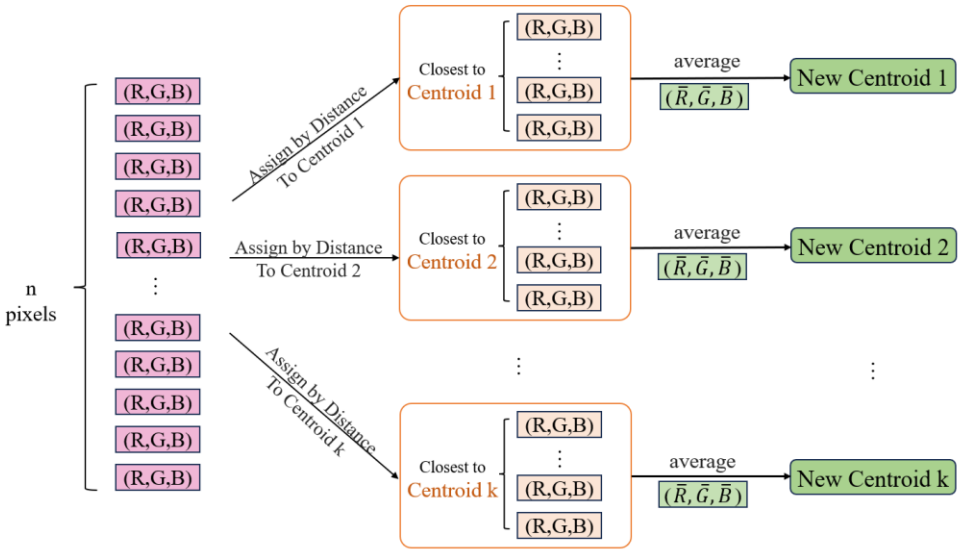

Fig 2.1 – Using k-Means to generate centroids

After getting the centroids, we still need to assign a centroid with the closest distance to each pixel and output them as the final generated image data.

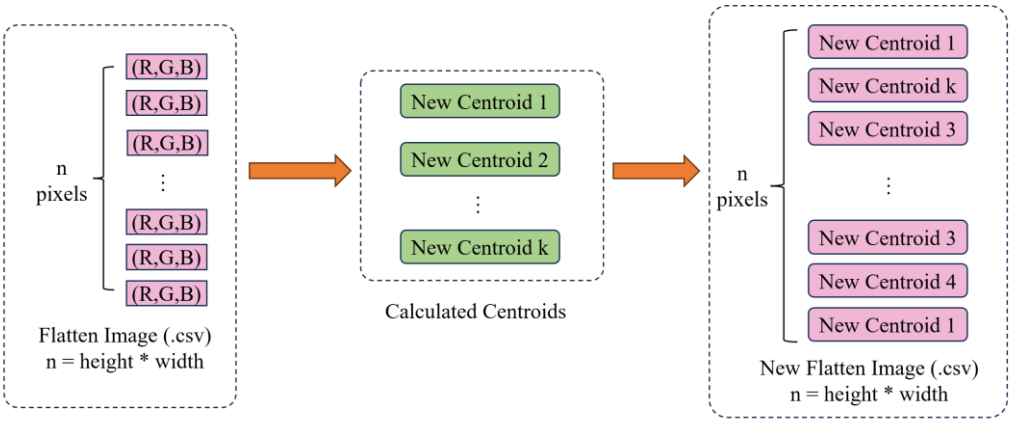

Fig 2.2 – Assign the nearest cluster centre to each pixel

For the Python baseline experiments, we did not use the scheme of multiple iterations of centroids due to the high time consumption. If the time taken for one iteration is t, based on our experiments, the time spent on one iteration exceeds 12 hours. Assuming that the average number of iterations for all images to converge is n, the expected time spent corresponds to approximately nt, which may take several weeks to compute. Therefore, we only consider one iteration of K-means here.

2.3 Time Cost for Baseline Python Experiment

Considering the time consumption, here we consider setting k to 32 for the time being, running one iteration for all the images and finally comparing the running time of all the experiments. For one round of iteration, when k is 32, it takes 43,455 seconds to process all images using baseline experiment, which is *more than 12 hours*. The time cost for images with different numbers of colours is shown in the figure below.

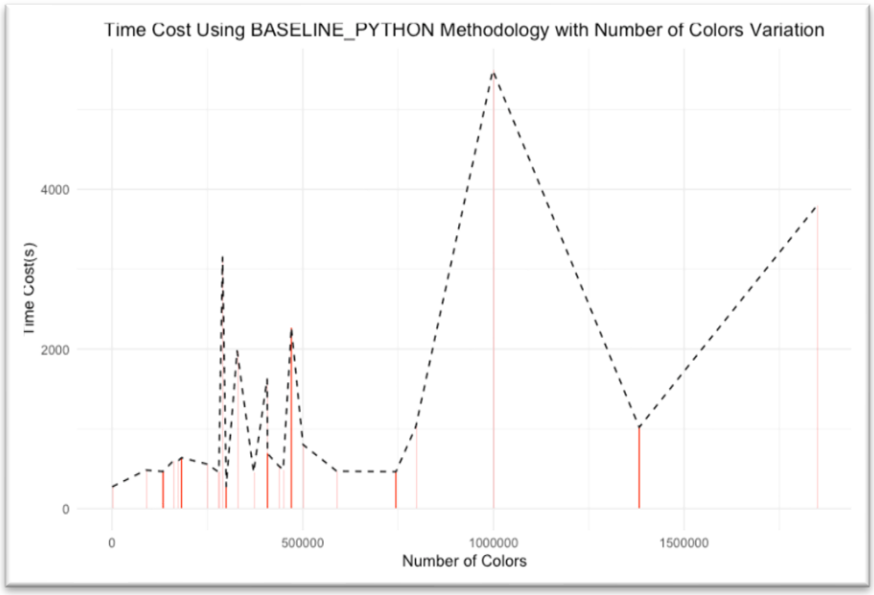

Fig 2.3 – Time cost comparison of images with different colors in *baseline Python* experiment

Here, only one of our result images is shown for visualizing the time cost of this experiment. For the other visualization images, we will put all the result of all the experiments in Chapter 6 with further analysis in Chapter 7.

3 Parallel Refactoring of Baseline Experiment using Joblib

The baseline experiment is executed sequentially, and it might be possible to speed up processing if it were refactored for parallel execution. Here, we use the Joblib package as our parallel processing tool.

Here are the two aspects that were optimized using Joblib library, including parallel align of nearest centroids and parallel updating of centroids.

3.1 Parallel Alignment of Nearest Centroids

The first operation that was optimized involves assigning each pixel in the image a label. This label is based on the pixel's proximity to the centres of certain clusters. Given the independent nature of this task for each pixel, it was an excellent candidate for parallelization. The following is the flowchart for parallel allocation of the nearest cluster centre.

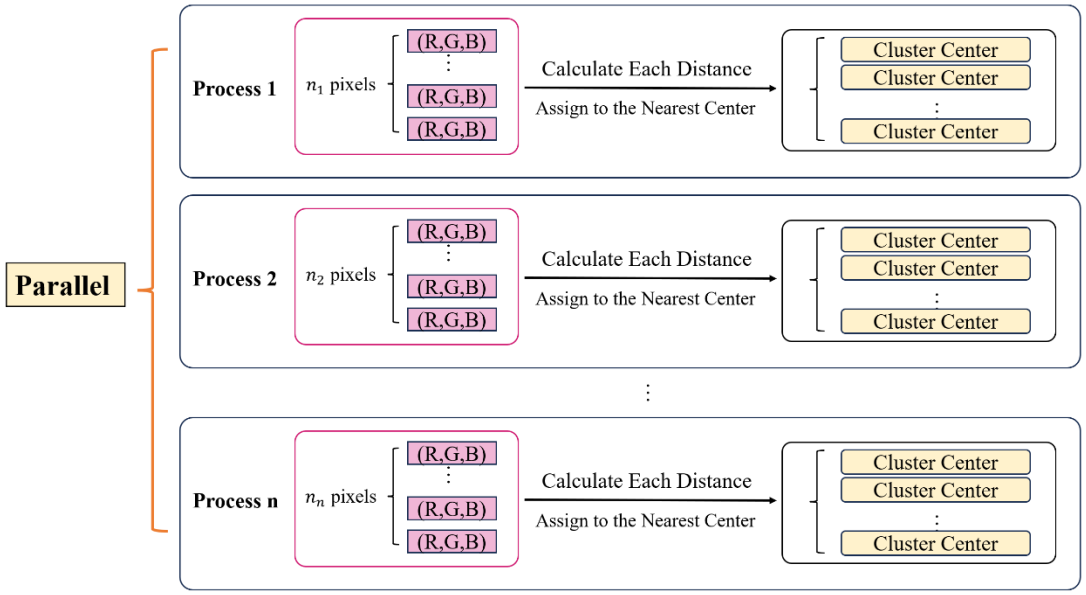

Fig 3.1 - Flowchart for parallel allocation of the nearest cluster center

Joblib will run multiple processes simultaneously, implementing a portion of the pixel data computation and processing on each process. Correspondingly, the code segment for allocating centroids refactored using Joblib is as follows:

Code Segment 1	**Assign to the Nearest Center**
labels = Parallel(n_jobs=os.cpu_count())(delayed(assign_label)(p, clusters) for p in tqdm(image))	

This method can be used here not only for categorizing each pixel according to the nearest centroid, but also for the final generation of image data.

3.2 Parallel Updating of Centroids

The second operation that was optimized involves updating the position of each cluster. This position is based on the average position of all pixels assigned to that cluster. Similar to the label assignment, these updates are independent for each cluster, making this operation another excellent candidate for parallelization. The following flowchart shows how this operation was parallelized.

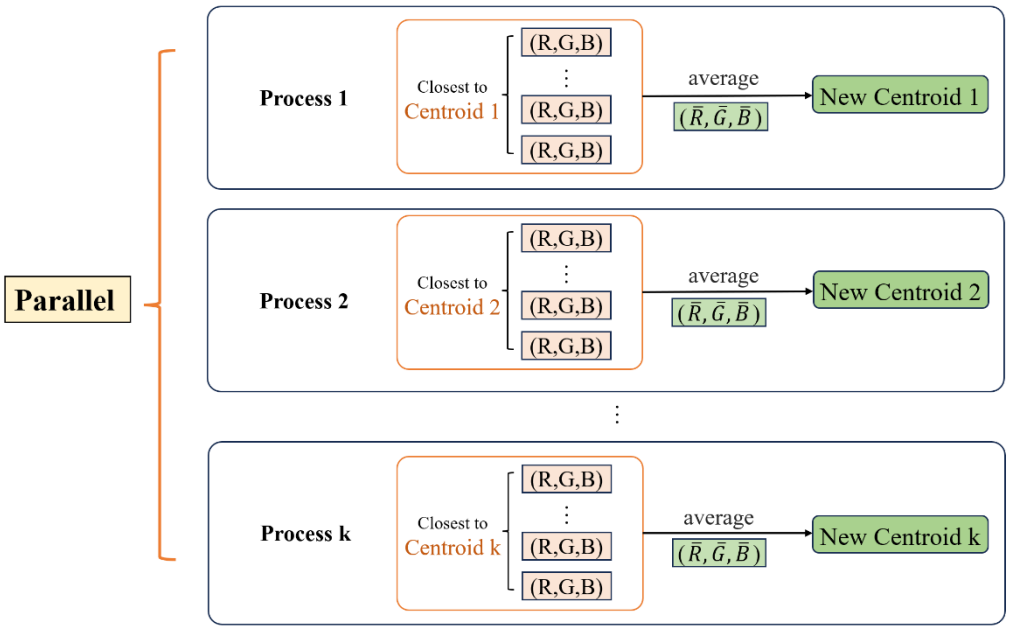

Fig 3.2 - Flowchart for parallel computing of new cluster centers

Similarly, Joblib runs multiple processes at the same time, implementing the computation of new centroids on each of them. The corresponding code segment using Joblib is as follows:

Code Segment 2	**Update Centroids**
clusters = Parallel(n_jobs=os.cpu_count())(delayed(update_cluster)(i, image, labels) for i in tqdm(range(K)))	

3.3 Time Cost for Parallel Python Experiment

Using Joblib's refactored parallel algorithm to process all images, when k=32, one iteration took a total of 34,332 seconds, which is *more than 9 and a half hours*, which saves about two and a half hours compared to the baseline experiment.

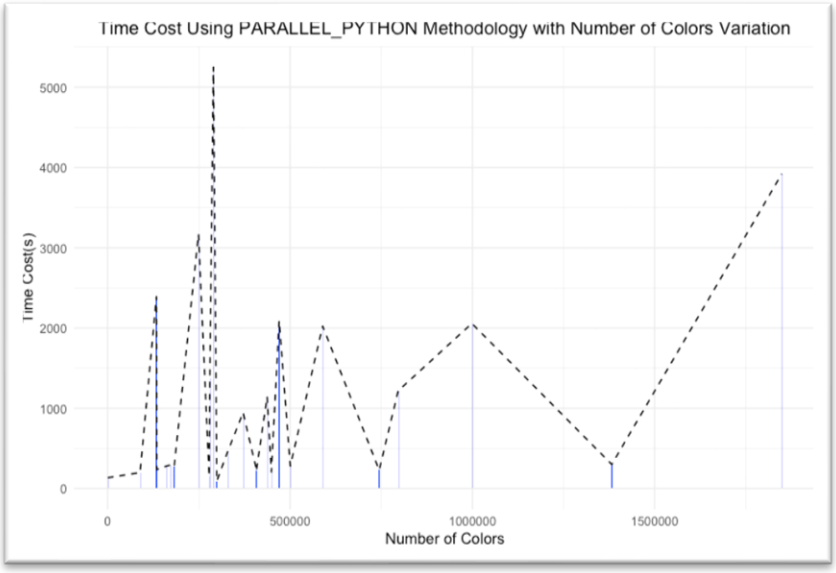

Fig 3.3 – Time cost comparison of images with different colours in *parallel Python* experiment

4 Parallel Implementation Based on Hadoop and Spark

In our previous experiments, we observed that refactoring with Joblib indeed significantly reduced the runtime, but the time cost was still substantial, with one single iteration also taking many hours. Is there a faster processing method? We can refactor our algorithm using Hadoop and Spark. Here, the experiment is divided into two parts. First, we use MapReduce to calculate the k centroids of the image. We can use the Hadoop MapReduce framework to split the data into multiple independent blocks for parallel processing. Since the Hadoop MapReduce framework cannot guarantee the order of data processing, we need another framework to assign each centroid to pixels in order. Here, we use Spark for processing, and in parallel, we assign the nearest centroid to each pixel in order, finally obtaining a flattened image represented by RGB triplets.

4.1 Calculate the Centroids based on Hadoop

The following picture show the MapReduce process and algorithm for computing Centroids.

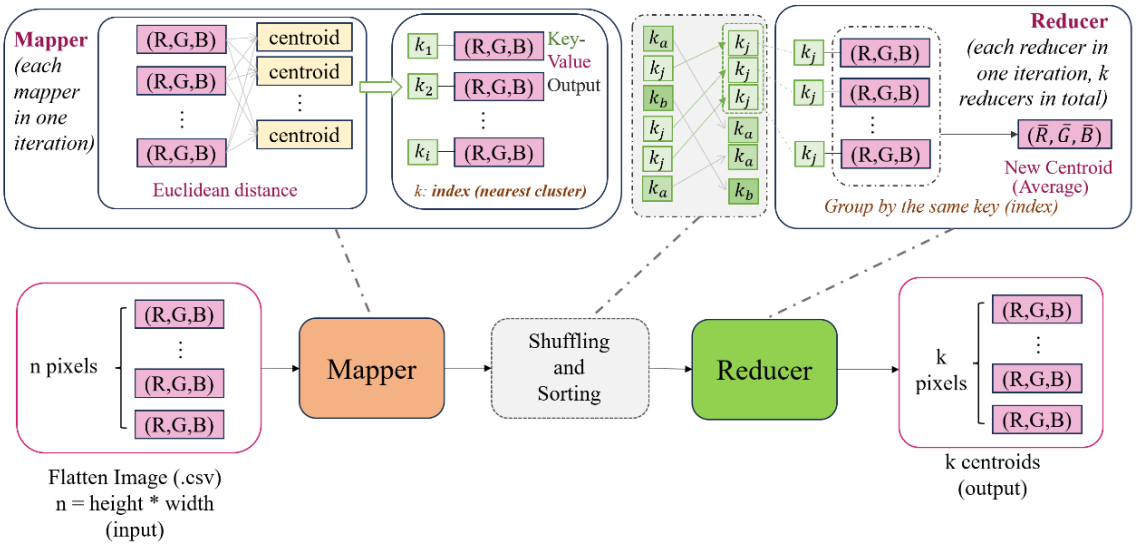

Fig 4.1 - Computing k centroids using MapReduce

To process large amounts of pixel data, the MapReduce framework randomly slices the pixel data into different splits and passes these slices to multiple Maps for processing. For the Mapper, we calculate the distance from each centroid for each pixel, use the label of the nearest centroid as the key, use the pixel itself as the value, and output this set of key-value pairs to the Reducer as an intermediate value. The pseudocode is as follows:

Algorithm 4-1 Mapper Class for k-Means MapReduce

Input: pixels, k

Output: key-value pairs as middle value, key is the nearest centroid label

```
1   Class KMeansMapper Extends Mapper
2       Function Setup(k)   <<Override>>   // generate or read centroids
3           If(first_iteration) Then
4               centroids = GenerateRandomCentroids(k)
5           Else Then
6               centroids = ReadCentroidsFromHadoopCache()
7           End If
8       End Function
9
10      Function Map(Key, Value, MapReduceContext)
11          Retrieve pixel from Value and, cluster from class
12          minDistance = INFINITE_VALUE
13          nearest_cluster = 0;
14          For i=0 to k Do
15              distance = calculateDistance(pixel)
16              If distance < minDistance Then
17                  minDistance = distance
18                  nearestCentroid= i
19              End If
20          End For
21          MapReduceContext.write(key= nearestCentroid, value=pixel)
22      End Function
23  End Class
```

Subsequently, the process enters a phase of Shuffling and Sorting. The MapReduce framework automatically sorts and categorizes the intermediate values according to their keys and assigns each category of data to the corresponding Reducer for processing. In our experiment, the key is the label of the centroid closest to the pixel, and the value is the RGB value of the pixel itself. Therefore, within each Reducer, we can calculate the average value of the set of pixels that are closest to the same centroid and output them as the new Centroid. The pseudocode of Reducer is as follows:

Algorithm 4-2 Reducer Class for k-Means MapReduce

Input: key-value pairs (middle value)

Output: key-value pairs as middle value, key is the nearest centroid label

```
1   Class KMeansReducer Extends Reducer
2       Function Reduce(Key, Value, MapReduceContext)
3           Retrieve pixels from Value
4           num = pixels.length // num of pixels
5           avg= (0,0,0)
6           For i=0 to num Do
7               For j=0 to 3 Do
8                   avg[j] = avg[j] + pixels[i][j]/num
9               End For
10          End For
11          MapReduceContext.write(key= NULL, value= avg)
12      End Function
13  End Class
```

We estimated that the time consumption of a single iteration is acceptable, therefore, we can consider implementing the multiple iterations. The flowchart is shown below:

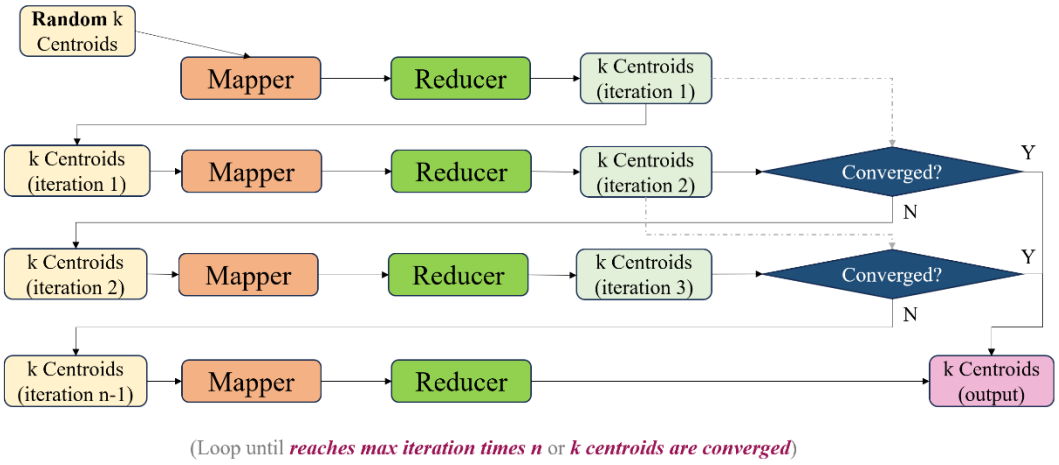

Fig 4.2 - Iterative Computation of k Centroids

Each iteration runs a separate MapReduce task. The result generated by each reducer (i.e., the centroids of each round) is compared with the centroids of the previous round to determine whether it has converged. If not, the new centroids will be passed to the mapper in the next round for a new iteration. This process will continue until convergence, or the maximum number of iterations is reached.

Algorithm 4-3 Determine whether the new centroids have converged

Input: max_iteration, k, image_csv_path

Output: k converged centroids as csv file

```
1   isDone = False
2   iteration = 0
3   While isDone is False AND iteration < max_iteration Do
4       Create Hadoop Configuration and Job
5       Set Mapper, Reducer and output Key-Value classes
6       Set input(image_csv_path) and output paths
7       If iteration > 0 Then
8           Add the previous centroids result to Job's cache files
9       End If
10      Wait for job to complete
11      If iteration > 0 Then
12          oldClusters, newClusters = read from cache files
13          isDone = isConverged(oldClusters, newClusters)
14      End If
15      iteration = iteration + 1
16  End While
```

In practical programming, we retain the centroids generated from each iteration. Starting from the second iteration, we store the centroids from the previous round in Hadoop's cache so that they can be read by the mapper when the job is running. The pseudocode for driving the MapReduce process is as follows:

4.2 Assign cluster centers to each pixel Based on Spark

Due to the inherent shuffling of original file order during Hadoop MapReduce operations, it is not feasible to directly use MapReduce for assigning nearest centroids to pixels in sequence. While MapReduce can ensure data order within a small partition, achieving global order requires the addition of unique indices to each pixel during the map phase. However, this approach increases storage requirements and can impact efficiency due to excessive indexing.

Utilizing Spark's Resilient Distributed Dataset (RDD) offers a more flexible and efficient solution for sequential data processing. Here is the process diagram for assigning centroids using Spark:

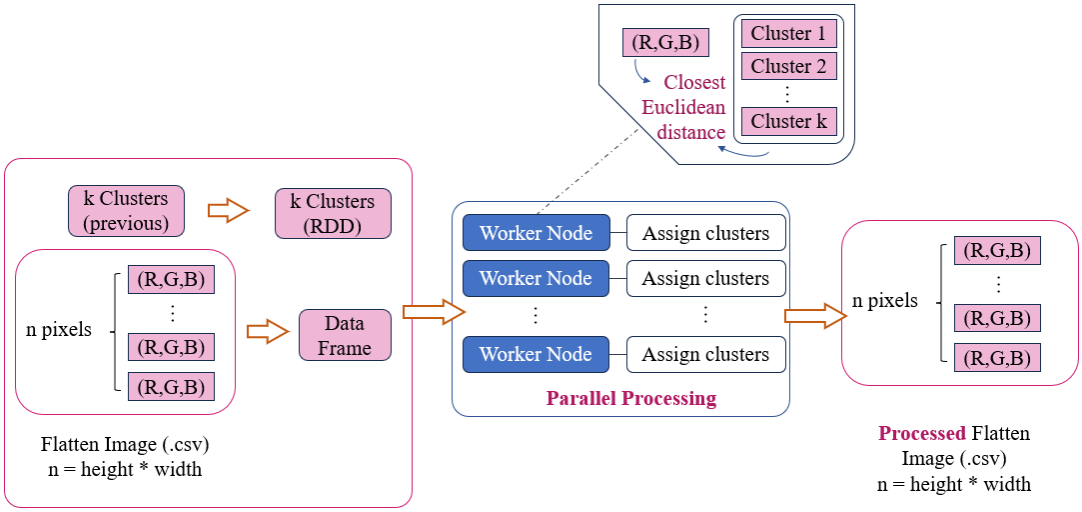

Fig 4.3 - Flowchart for parallel computing of new cluster centers using Spark

4.3 Time Cost for Hadoop with Spark Experiment

For this experiment, we counted the time spent calculating centroids and allocating centroids separately and added them together to calculate the total time. The time taken for one iteration when k=32 is 1,393 seconds (about 23 minutes), which is *only 23.22 minutes*. This experiment greatly reduced the time cost. The time cost of 1 iteration for images with different numbers of colors is shown in the figure below.

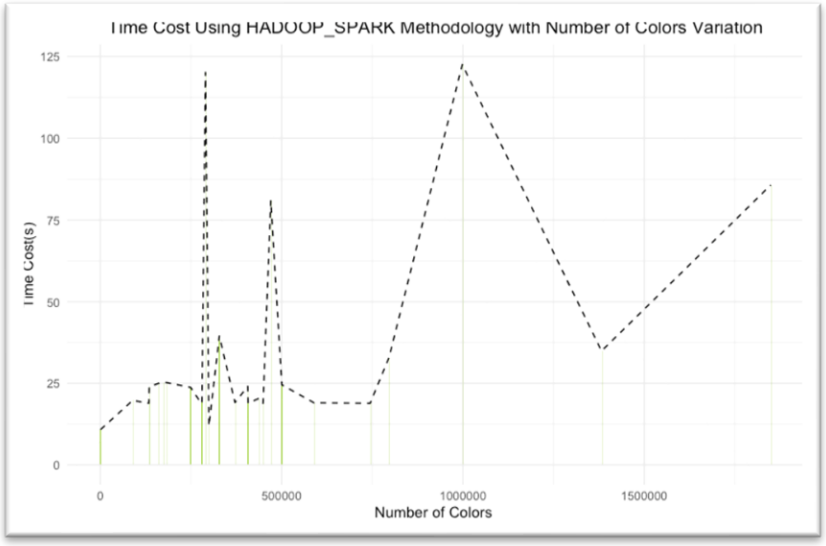

Fig 4.4 – Time cost comparison of images with different colors in Hadoop with Spark experiment

5 Parallel Implementation Based on Pure Spark

5.1 Methodology of Parallel Computing using Spark

In chapter 4, we explored parallel image compression implementation that combined Hadoop and Spark. Hadoop, as a framework capable of processing large datasets, performs well in our image processing tasks. However, with Spark's advantages in in-memory computation, we decided to put efforts on a pure Spark implementation, which we elaborate on in this chapter. Spark not only inherits the distributed storage features of Hadoop but also enhances processing speed, which is crucial for large-scale and real-time image processing applications.

Following the advantages of Hadoop's distributed file system, the choice of Spark was driven by the pursuit of higher computational efficiency and speed. Spark's Resilient Distributed Datasets (RDDs) and in-memory computing make it suitable for image compression tasks. Moreover, compared to Hadoop, Spark provides a faster data processing pipeline, enabling us to execute more efficient compression algorithms.

The workflow of this experiment is shown below.

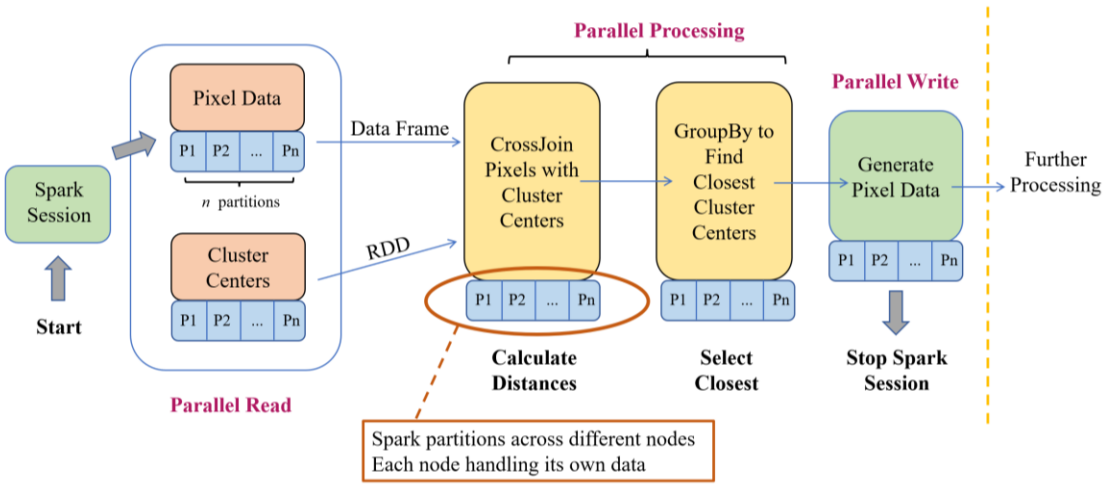

Fig 5.1 - The Workflow of Distributed Color Quantization Using Pure Spark

The operation commences with a Spark session. We firstly load pixel data into Spark DataFrames, which provides an efficient way to operate on and optimize tabular data. Then, we convert cluster center data into RDDs, the core of Spark that is designed for parallel processing. Our code iterates, governed by a max_iteration limit, refining cluster centroids via a MapReduce job. Only when centroids stabilize does the iteration halt, outputting the final cluster centroids in a CSV file.

For data processing, pixels are matched to centroids, distances calculated, and clusters formed—all in parallel. This step is crucial in our compression algorithm, as it allows us to map the original pixel colors to the colors of the cluster centers, thus reducing the

total number of colors and compressing the image. We find the nearest cluster centers for each pixel in parallel. After determining the nearest cluster centers, we generate new pixel data that represents the compressed image, and the Spark session closes.

Leveraging Hadoop and Spark, the algorithm efficiently clusters large-scale data, proving ideal for image compression. The process is iterative, yet efficient, ensuring scalability and performance for big data challenges.

Algorithm 5-1 Pixel Cluster Compression with RDD

Input: original pixel csv file

Output: new pixel csv file with segmented image data using color quantization

```
1   Function ColorQuantization (original pixel csv file, cluster file)
2       Randomly select initial cluster centers from pixel vectors
3       Initialize convergence, iteration counter, max_iteration and threshold
4       While NOT converged AND iteration counter < max_iteration Do
5           For each pixel vector in RDD
6               Compute distance to all cluster centers & assign to closest center
7           End For
8           Group pixel vectors by assigned cluster center
9           Compute new cluster centers as average of each group
10          If change in centers < threshold Then
11              convergence = TRUE
12          End If
13          Increment iteration counter
14      End While
15      Map each pixel vector in RDD to its closest cluster center
16  End Function
```

5.2 Time Cost for Pure Spark Experiment

For this experiment, one iteration took 979 seconds at k=32, which is only *16.31 minutes*. The time cost of pure spark experiments is lower than that of experiments combining Hadoop and Spark. Here is the figure showing the time cost of 1 iteration for images with different numbers of colors:

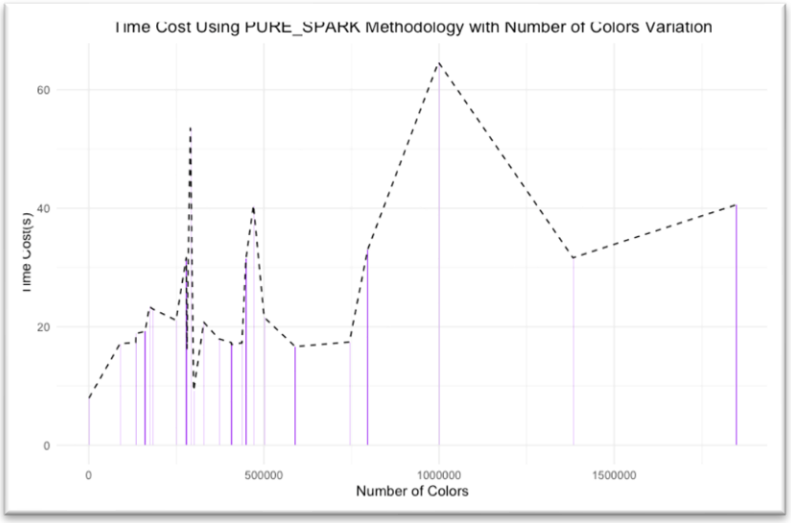

Fig 5.2 – Time cost comparison of images with different colors in pure Spark experiment

6 Results and Conclusions

We conduct a comprehensive analysis of our experimental outcomes by meticulously comparing the processing duration required for the dataset and the specific alterations in image attributes. This comparison is facilitated through the application of four distinct image quantization methodologies, each offering a unique approach to image color quantization. Our statistical comparison focuses on quantifying the efficiency of each method in terms of time efficiency and the extent of changes they induce in key image properties such as colour and size.

6.1 Time Cost Comparisons

We calculated the time required to compress the same image using different methodologies and different values of K ranging from 2 to 256. Based on these data, we created a comparison chart as shown in Figure 6.1

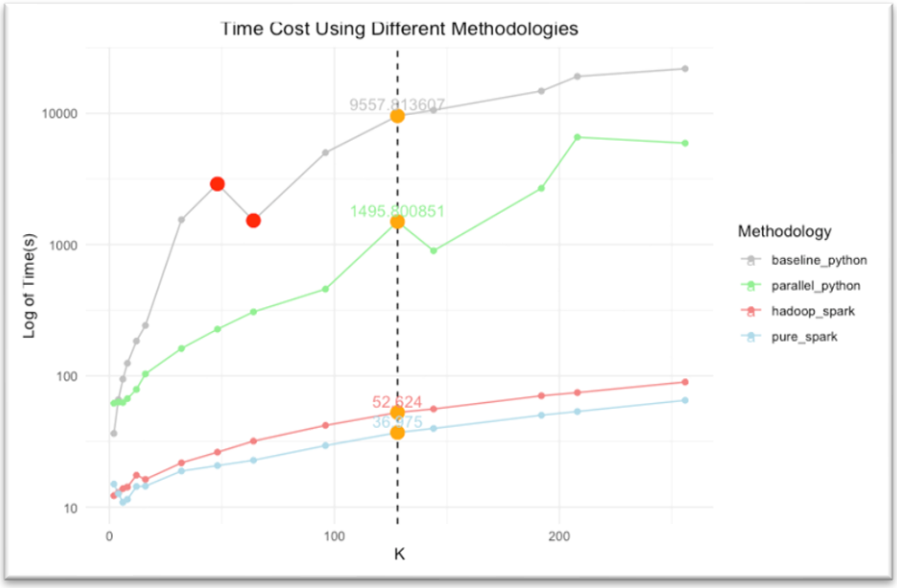

Fig 6.1-Time cost using different methodologies

We can observe that the total time needed to process this image using four different methodologies varies significantly. Based on the given image, it can be observed that there is a positive correlation between the time cost and the value of ‘k’ for the same image. In other words, the higher the ‘k’ value, the greater the time cost to process an image. However, this positive correlation does not necessarily imply that an increase in ‘k’ will invariably lead to an increase in time cost (as exemplified by the red points in the figure). We hypothesize that this may be related to the random selection of initial clusters and the CPU status of the computer during operation.

Among the four experiments, the use of Hadoop and Spark significantly reduced the time cost. The experiment using pure Spark had a slightly lower time cost than the experiment that combined Hadoop and Spark, but the difference was not as pronounced

as the gap between them and the two Python experiments.

We used four different methods (with 'K' set to 32) to calculate the time required to process the same image with two different color bits (8-bit and 16-bit respectively), as shown in Figure 6.2.

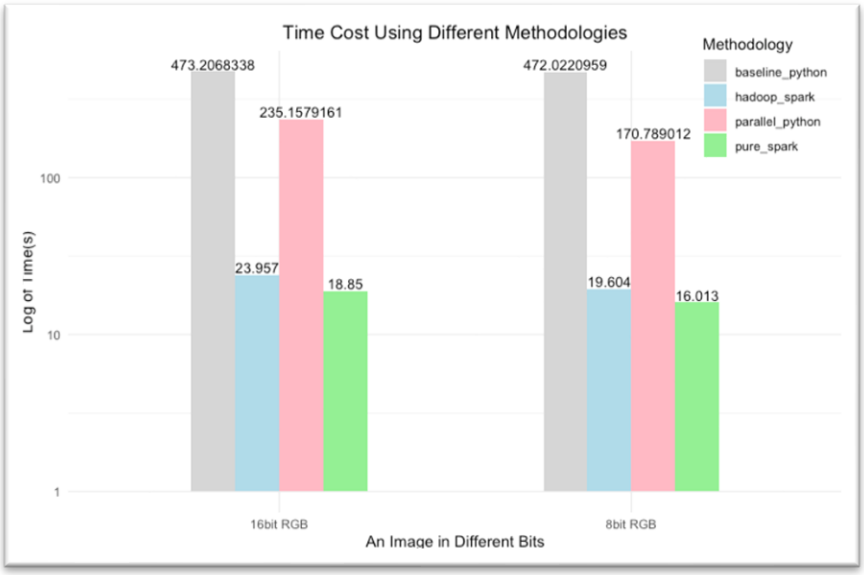

Fig 6.2 -Time cost using different methodologies

As can be seen, the time for color quantization of 8-bit images is indeed slightly lower than that of 16-bit images. However, compared to the differences between different experimental methods, the difference is not significant. Therefore, we believe that the main source of time cost differences lies in the different parallel implementations.

We calculated the total time taken to process two different sets of data (8bit and 16bit respectively) using four different methodologies (K is 32) in Figure 6.3.

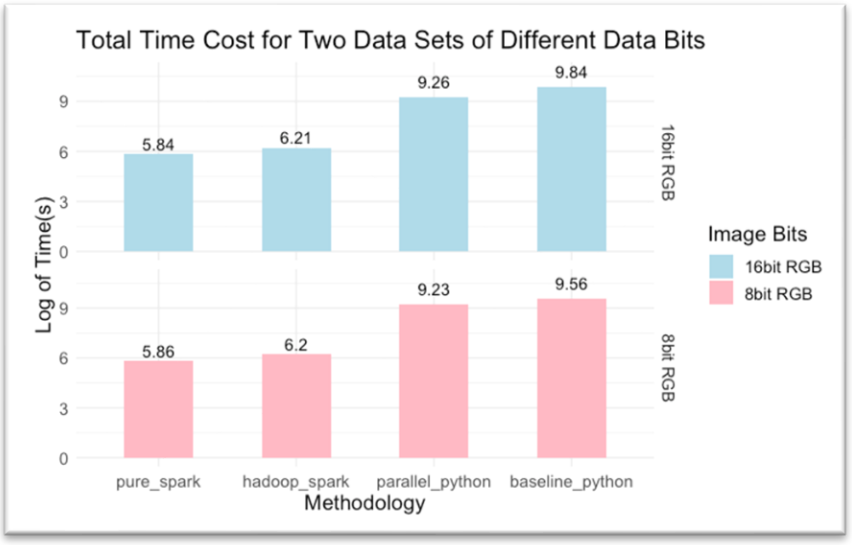

Fig 6.3 - Total time cost for two data sets of different data bits

The conclusion here is the same as the previous one. The differences of runtime come from different parallel algorithms, while the color bits have a subtle effect.

We compare the effect of the number of pixels in an image on the running time. This result figure shown in the figure below (k=32):

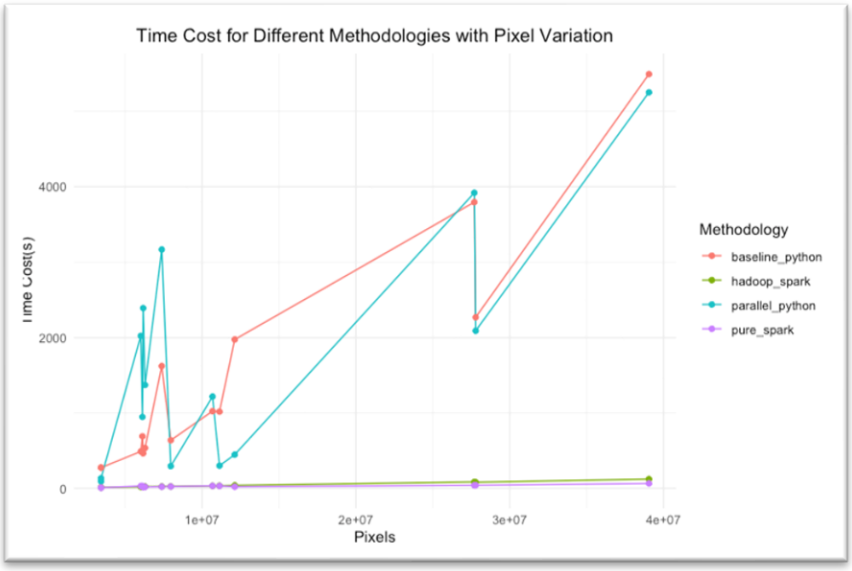

Fig 6.4 - Time cost for different methodologies with pixel variation

As observed, the number of pixels in an image generally correlates positively with runtime, with certain outliers potentially attributable to image content and structure. Furthermore, we investigated the influence of color count on runtime. We processed images with varying color counts using four methods (K=32), as depicted in Figure 6.5.

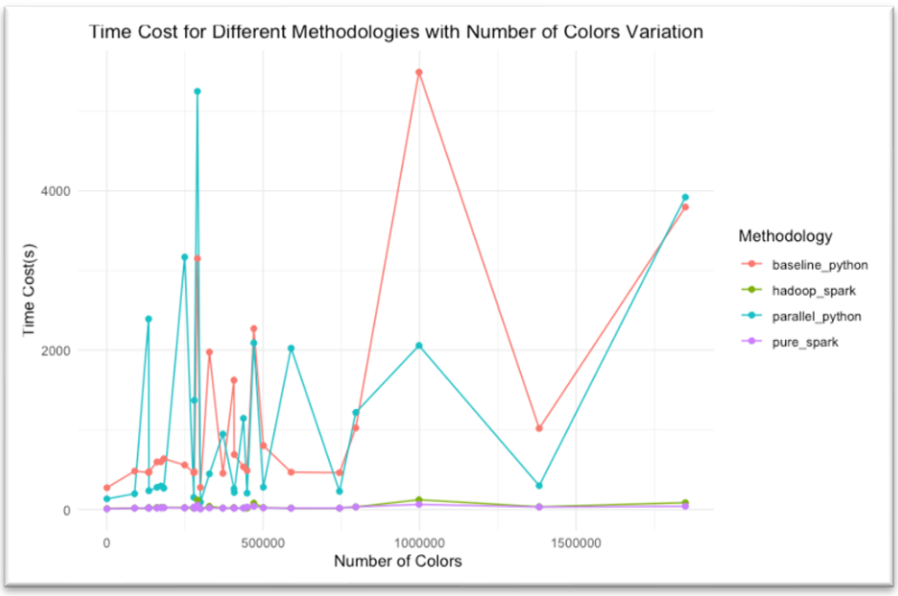

Fig 6.5 - Time cost for different methodologies with number of colors variation

We can see that the effect of different number of colors on the running time is irregular,

which may be related to the structure and content of the image itself, but we can vaguely see a positive correlation between them. In the image, we can see sudden spikes in outliers (for example, when the image has 100M colors). These outliers cause significant disturbances in the Python experiments, but only minor disturbances in Hadoop and Spark. We can assert that Hadoop and Spark demonstrate robustness in terms of runtime.

6.2 Effect of Color Quantization on Image Properties

Based on the above analysis of the runtime, we can easily get the conclusion that the runtime can be significantly affected by the parallel algorithm. Next, we would like to explore the effect of color quantization algorithms on the image itself, including image quality, size, etc.

We converted the image data, which had undergone color quantization, back into PNG format for comparison purposes. The original image was also saved in PNG format. Taking a photograph as an example, the size of the image saved in PNG format without undergoing color quantization is 1748.16KB. After color quantization with different *k* values (i.e., representing an image with different numbers of colors), the sizes of the images saved in PNG format are compared as shown in the following figure:

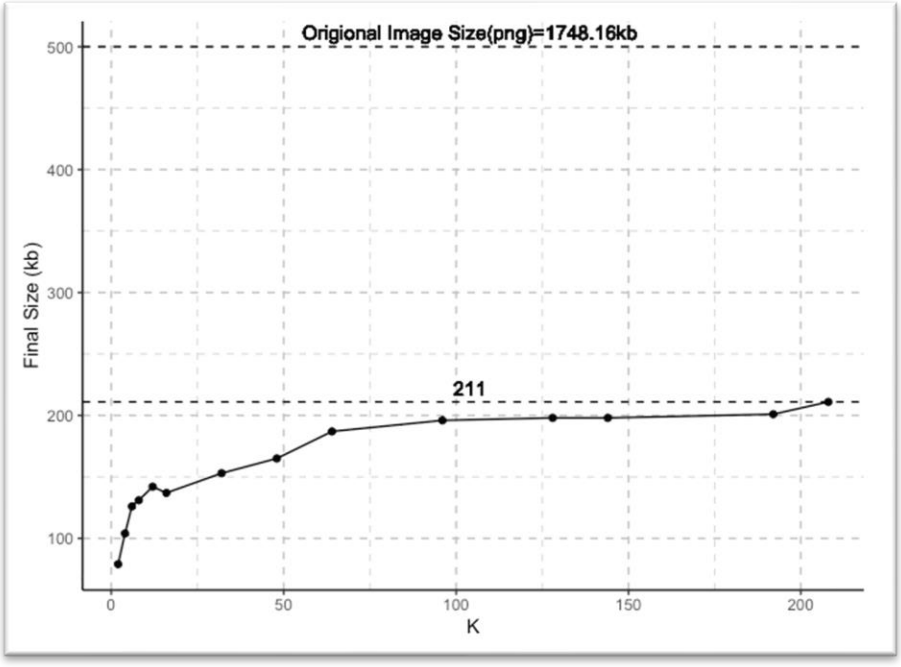

Fig 6.6 - The size of one image after color quantization with different k

After undergoing color quantization segmentation, the size of the image exhibits a positive correlation with the *k* value. In our experiments, the *k* value ranged from 2 to 256, and the maximum size of the processed image was less than 211KB. The size of the quantized image was only 12.07% or less of the original image, indicating that this process reduced approximately 87.93% or more of the space. This suggests that color

quantization not only effectively shortens the processing time but also significantly reduces the volume, implying that color quantization can achieve image compression to a certain extent.

Next we look at the image itself after color quantization.

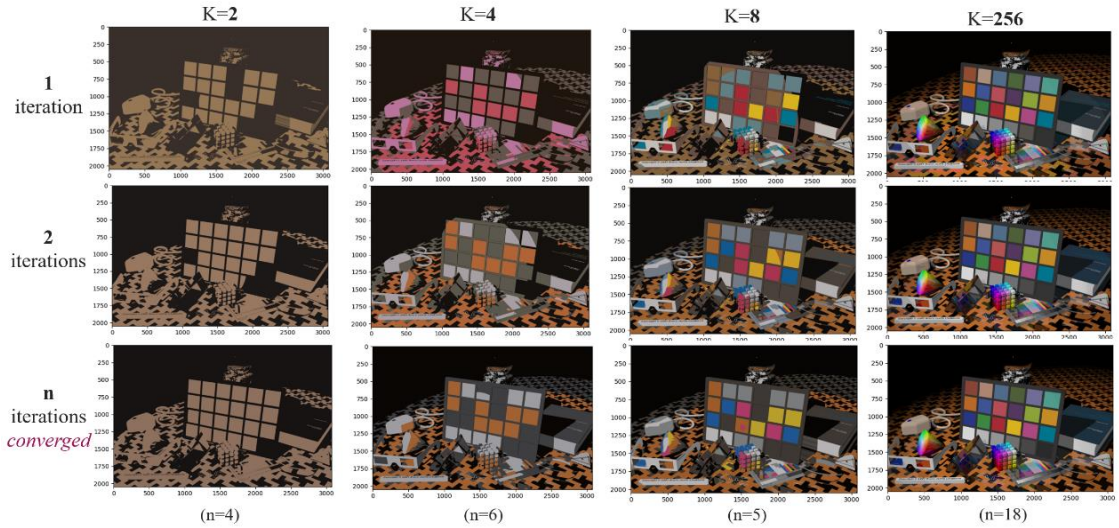

Fig 6.7 - Images after color quantization for different k and number of iterations

For the first iteration, colors are less accurate. For instance, when *k* equals 4, the image appears pinkish, possibly due to the random centroids leaning towards pink. From the second iteration, image quality improves significantly. More iterations and larger 'k' values enhance color accuracy and resemblance of the quantized image to the original.

The number *N* of iterations until convergence may be related to *k*. Therefore, we also recorded the number of iterations required for the centroids to converge under different *k* values.

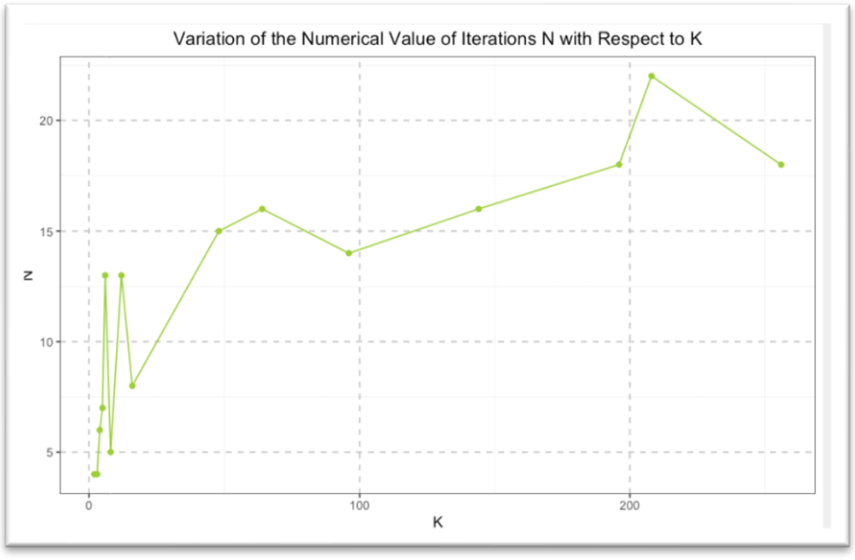

Fig 6.8 - Variation of the number of iterations *N* with different *k*

Based on the image, we can observe a general trend that a larger k indeed leads to a greater number N of iterations until convergence. Moreover, the results may vary depending on the selection of initial centroids. We can further infer that in order to obtain a more accurate image after color quantization, its computational complexity is not only related to the k value in a single iteration, but also to the number of iterations influenced by the k value.

6.3 Conclusion

Based on the analysis of the experimental results, we can summarize the conclusions here.

The primary factors influencing runtime include the parallel algorithm and the 'k' value. As the 'k' value increases, the time cost of processing the image also increases, but this is not an absolute correlation. The use of Hadoop and Spark significantly reduces the time cost, with the pure Spark experiment exhibiting a slightly lower time cost than the experiment combining Hadoop and Spark. Additionally, the number of pixels and the number of colors in the image also affect the runtime, but their impact is relatively minor. Our experiments demonstrate that Hadoop and Spark exhibit robustness in terms of runtime.

Color quantization not only effectively shortens processing time but also significantly reduces the size of the image, thereby achieving a degree of image compression. The color accuracy of the image and its resemblance to the original can be enhanced by increasing the number of iterations and enlarging the k value. Furthermore, the computational complexity of obtaining a more accurate image after color quantization is related not only to the k value in a single iteration but also to the number of iterations influenced by the k value. These factors collectively impact the quality and size of the image.

7 Further Discussion

7.1 Discussion on Our Results and Conclusions

Specific image color quantization may take into account other more factors depending on the usage scenario. There are still some factors that are not taken into account in our experiments. First, for the algorithm that determines the convergence of centroids, we need to specify a threshold. This one threshold may affect the number of iterations and thus the quality of the image and the running time.

Due to our limited equipment, we did not test the performance of these algorithms on other hardware platforms and operating systems. Using computers with more memory, clustered servers, or CUDA-based utilizing GPU acceleration may have better performance.

But all in all, we still think that platforms using Hadoop and Spark are effective, both of them can generate quantized images correctly, and both of them can ensure that the running time of the images is within a certain acceptable range.

7.2 Potential Directions for Future Research

In our experiments, we have explored only one method for performing color quantization, which is kMeans. But in fact, there are many other ways to perform color quantization[2], and perhaps some of them can be rewritten for parallel computation and may perform well.

In our experiments, in order to generate an image, we need to prespecify the number of colors *k*. Although we discuss the impact of the value of *k*, however, in some applications, determining the value of *k* is an important issue[6,7]. Therefore, it might be possible to design an algorithm for images that automatically finds a suitable *k*.

8 References

[1] Braquelaire, J. P., & Brun, L. (1997). Comparison and optimization of methods of color image quantization. IEEE transactions on image processing:a publication of the IEEE Signal Processing Society, 6(7), 1048–1052. https://doi.org/10.1109/83.597280

[2] Celebi, M. Emre. (2023). Forty years of color quantization: a modern, algorithmic survey. Artificial Intelligence Review. 56. 1-82. 10.1007/s10462-023-10406-6

[3] Tirandaz, Z., Foster, D.H., Romero, J. et al. (2023). Efficient quantization of painting images by relevant colors. Sci Rep 13, 3034. https://doi.org/10.1038/s41598-023-29380-8

[4] Su-yu Huang and Bo Zhang. (2021). Research and improvement of k-means parallel multi-association clustering algorithm. In Proceedings of the 2020 International Conference on Cyberspace Innovation of Advanced Technologies (CIAT 2020). Association for Computing Machinery, New York, NY, USA, 164–168. https://doi.org/10.1145/3444370.3444565

[5] Rawzor. “The New Test Images - Image Compression Benchmark,”. 2023. [Online]. Available: https://imagecompression.info/test_images/. [Accessed: Dec. 2, 2023].

[6] Celebi ME, Wen Q, Hwang S (2015) An effective real-time color quantization method based on divisive hierarchical clustering. J Real-Time Image Proc 10(2):329–344

[7] Barata C, Celebi ME, Marques JS et al (2016) Clinically inspired analysis of dermoscopy images using a generative model. Comput Vis Image Underst 151:124–137

9 Individual Contribution

[Hidden]

10 Dataset, Code and Result Availability

The original dataset can be downloaded at the following URL:
https://imagecompression.info/test_images/

We selected RGB images from the original dataset. For the dataset we used, it is available for download at the link below:
https://drive.google.com/file/d/1P0heLnF67NreUprIO2VLmf_zqRCxNaJ6/view?usp=sharing

We converted these images to csv format and they were decompressed to 4.05 GB in size and these processed images in csv format can be accessed in Google Drive at the following URL:
https://drive.google.com/file/d/1McngLH9i6VASdI9iGaCBjeLOTfYkceQ-/view?usp=drive_link

We publicly uploaded both our code and results to GitHub, the link to the GitHub repository is shown below:
https://github.com/YongzeYang/Parallel-Color-Quantization/tree/main

--- END ---