COMP 605 Project: Parallel Image Segmentation K-Means Clustering and Otsu's Method

Luna Huynh and Daisy Ulloa

May 1, 2024

1 Introduction

Image segmentation is a technique in image processing and computer vision where a digital image is partitioned into multiple segments (discrete groups of pixels). The purpose of segmentation is to simplify or change the representation of an image into something that is more meaningful and easier for the computer systems to analyze [1]. The goal of this project is to implement two image segmentation methods, K-Means clustering and Otsu's Method in C using two of the most widely used application programming interfaces (APIs) for parallel programming, OpenMP and MPI. Otsu's method is specified to searching for the global optimal threshold, while K-Means offers more flexibility and can find a locally optimal solution [4]. We first introduce the algorithm of K-Means Clustering Method and Otsu's Method in image segmentation. We then present test images, the details of how we parallelized these two algorithms. Finally, we present the resulted images obtained from our code, the speedup of K-Means and Otsu with OpenMP and MPI, as well as the recommended number of threads for different image sizes of the test image. Our results show Otsu is a more reliable method, whereas K-Means performs better than Otsu for smaller amount of processors. We also conclude that OpenMP is easier to work with and for this project, performs well. However, MPI is overall faster than OpenMP.

2 Algorithms

2.1 K-Means Clustering

K-Means is a popular choice for image segmentation due to its ability to segment an image to as many color regions as the user would like. K-Means is a clustering algorithm that assigns pixels in an image to a defined cluster point with a specific color value. Once pixels are assigned, the cluster point is centered to be the optimal color value of its cluster(average color). Pixels in the image are assigned once again and the process repeats until the optimal clustering is found. The pixels are then recolored to the value of the cluster point. The full sequential process is shown in Figure 1 [2].

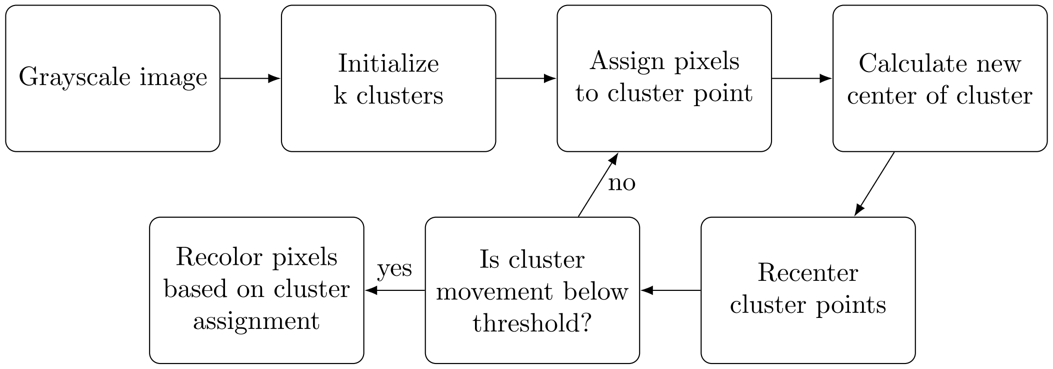

Grayscale image
Initialize k clusters
Assign pixels to cluster point
Calculate new center of cluster
no
Recolor pixels based on cluster assignment
yes
Is cluster movement below threshold?
Recenter cluster points

Figure 1: Flowchart of sequential K-Means Method

For grayscale images, the pixel color values range from 0 to 255. Each pixel is denoted as x_i. The following are the mathematical steps for K-Means [3].

1. Define threshold value for re-centering clusters
2. Assign k clusters with random color value from 0 to 255.
3. For each pixel x_i in the image, compute the distance d to each cluster k_j.

$$d_{ij} = |x_i - k_j| \tag{1}$$

4. Calculate the mean or center of the cluster

$$k_j^* = \frac{1}{c_j} \sum_{i=1}^{c_j} x_{ij} \tag{2}$$

   where c_j is the count of pixels in cluster k_j and x_{ij} belongs to cluster k_j.

5. Check if $|k_j - k_j^*| <$ threshold.
   (a) If difference is below threshold, continue.
   (b) If difference is above threshold, repeat steps 3, 4, and 5.
6. Change color value of all x_{ij} belonging to cluster k_j to defined values.

2.2 Otsu's Method

Otsu's Method is one of the most widely known thresholding technique that is designed for determining the optimal threshold grayscale intensity by separating image pixels into two classes foreground and background. This method is particularly effective as it is automatically searching for the global optimal threshold by maximizing the between-class variance. The following flowchart illustrates the sequential steps involved in implementing Otsu's Method as well as provides a brief understanding of the algorithm's operation, from initial grayscale to the final segmented image output.

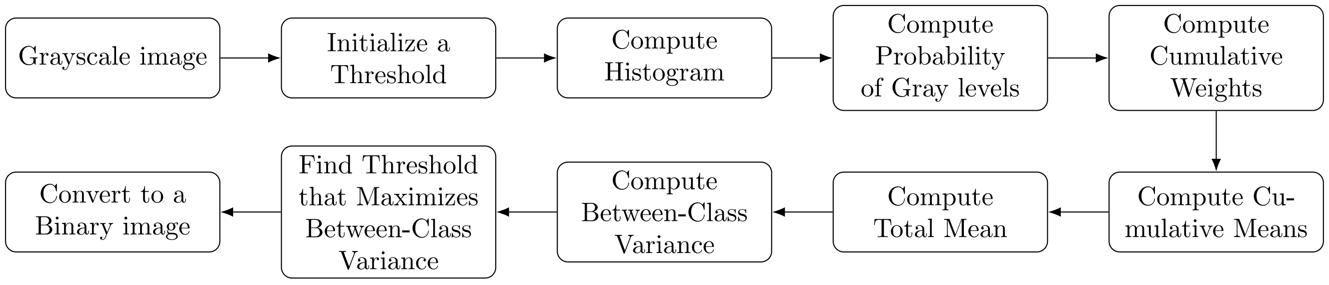

Grayscale image
Initialize a Threshold
Compute Histogram
Compute Probability of Gray levels
Compute Cumulative Weights
Compute Cumulative Means
Compute Total Mean
Compute Between-Class Variance
Find Threshold that Maximizes Between-Class Variance
Convert to a Binary image

Figure 2: Flowchart of sequential Otsu's Method

Assuming an image is represented in L gray levels from 0 to $L-1$. The number of pixels at level i is denoted by n_i. The total number of pixels is denoted $N = n_1 + n_2 + ... + n_L$. The sequential Otsu algorithm implemented for grayscale images consists of following steps [4]:

1. Store the histogram with all possible gray levels
2. Compute the probability of each intensity level i

$$p_i = \frac{n_i}{N}, \quad p_i \geq 0, \quad \sum_{i=0}^{L-1} p_i = 1 \tag{3}$$

3. Initialize a threshold t
4. The gray level probability distribution is divided into two classes:

   Background with gray levels $[0, 1, ..., t]$

   Foreground with gray levels $[t+1, ..., L-1]$
5. Compute the cumulative weights for each class

$$w_B = \sum_{i=0}^{t} p_i \tag{4}$$

$$w_F = \sum_{i=t+1}^{L-1} p_i \tag{5}$$

6. Compute the cumulative means for each class

$$\mu_B = \sum_{i=0}^{t} i \left(\frac{p_i}{w_B}\right) \tag{6}$$

$$\mu_F = \sum_{i=t+1}^{L-1} i \left(\frac{p_i}{w_F}\right) \tag{7}$$

7. Compute the total mean of gray levels

$$\mu_T = w_B \mu_B + w_F \mu_F \tag{8}$$

8. For each threshold value, calculate the between-class variance.

$$\sigma_{bc}^2 = w_B(\mu_B - \mu_T)^2 + w_F(\mu_F - \mu_T)^2 \tag{9}$$

9. Determine the optimal threshold value t by maximizing the between-class variance.
10. Change foreground pixels to black by replacing pixel values with $L-1$ and background pixels to white by replacing pixel values with 0
11. Convert to a binary image

3 Implementation

3.1 Test Images

We segment 8-bit grayscale images with the total number of gray levels of 256 in three different sizes 128x128 pixels, 256x256 pixels, 512x512 pixels. The range of the intensity values is between 0 and 255.

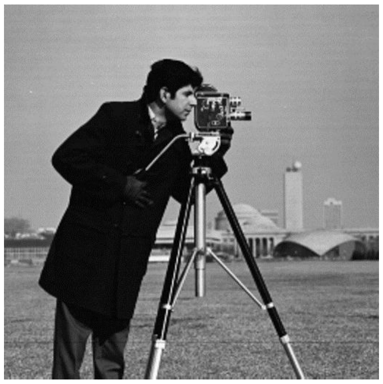


Figure 3: Original 512x512 grayscale image

3.2 K-Means

To compare K-Means to Otsu as best we could, we recolor the pixels in the image to be either black and white, compared to the average color of the cluster that is typical in K-Means examples. This added an extra step of changing the final k_j centers to be either 0 or 255 depending on which is smaller and larger in color value. The image is then recolored.

Figure 4 displays which steps were parallelized in OpenMP and MPI.

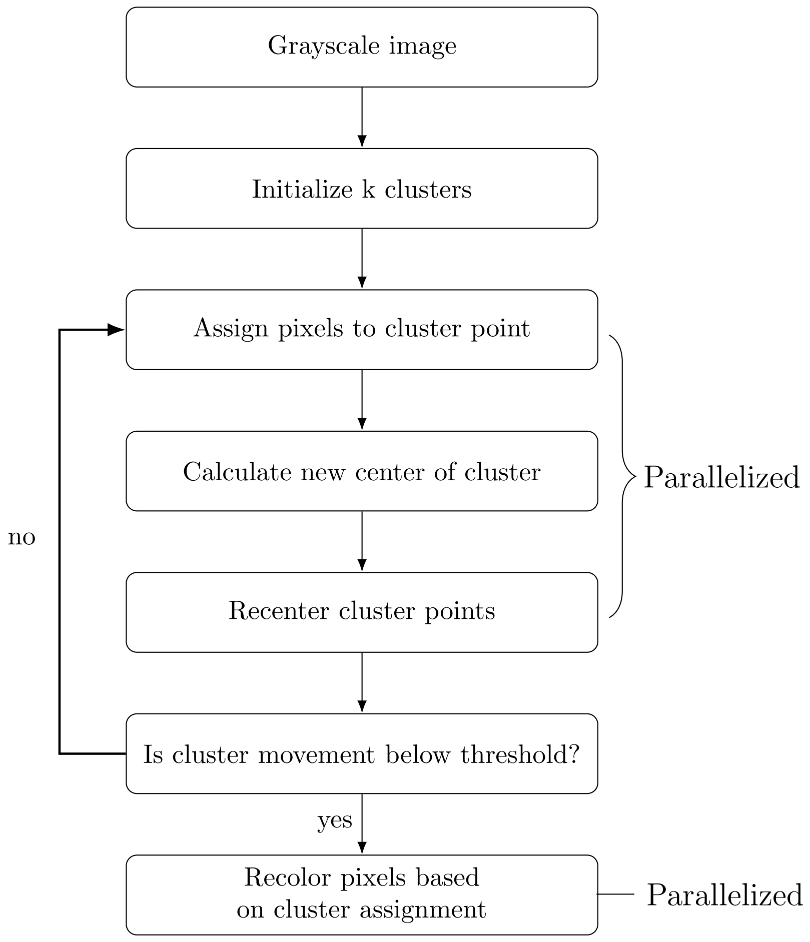

Grayscale image
Initialize k clusters
Assign pixels to cluster point
Calculate new center of cluster
Recenter cluster points
Is cluster movement below threshold?
no
yes
Recolor pixels based on cluster assignment
Parallelized
Parallelized

Figure 4: Flowchart of sequential K-Means Method

3.3 Otsu's Method

The following flowchart visually emphasizes the steps in Otsu's Method that can be parallelized in OpenMP and MPI.

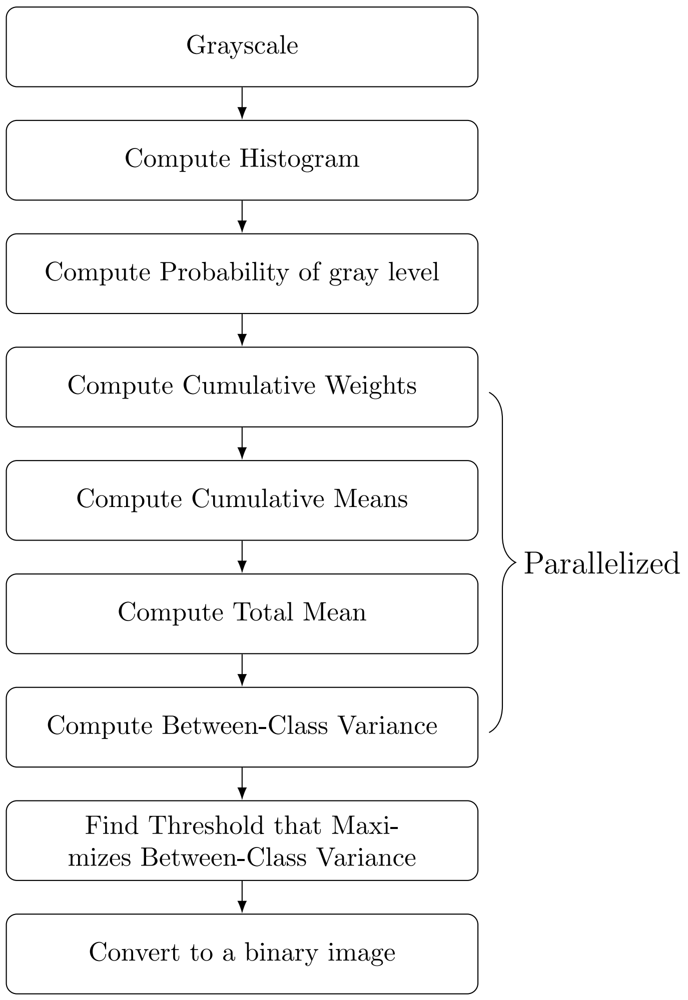

Grayscale
Compute Histogram
Compute Probability of gray level
Compute Cumulative Weights
Compute Cumulative Means
Compute Total Mean
Compute Between-Class Variance
Parallelized
Find Threshold that Maximizes Between-Class Variance
Convert to a binary image

Figure 5: Flowchart of parallel Otsu's Method

4 Results

4.1 Segmented Images

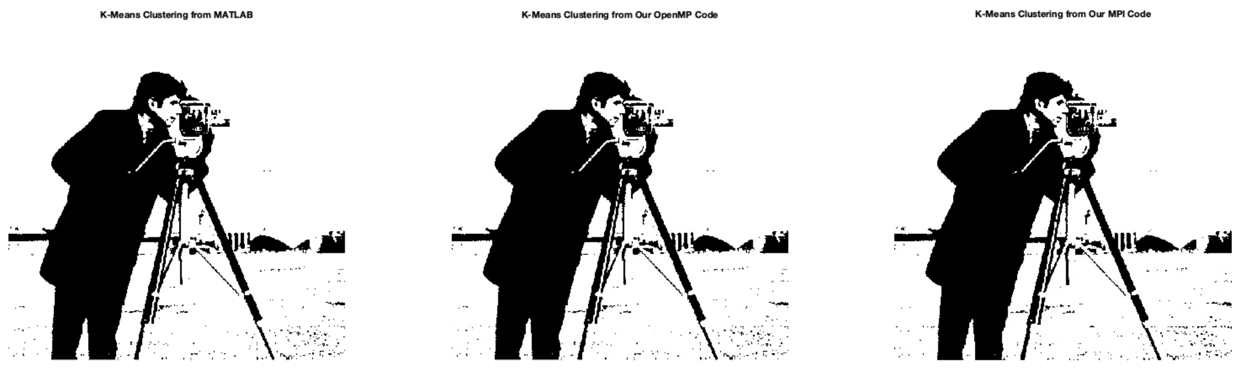

K-Means Clustering from MATLAB
K-Means Clustering from Our OpenMP Code
K-Means Clustering from Our MPI Code

Figure 6: Segmented 512x512 images with K-Means using MATLAB (*left*), OpenMPI (*center*) and MPI (*right*)

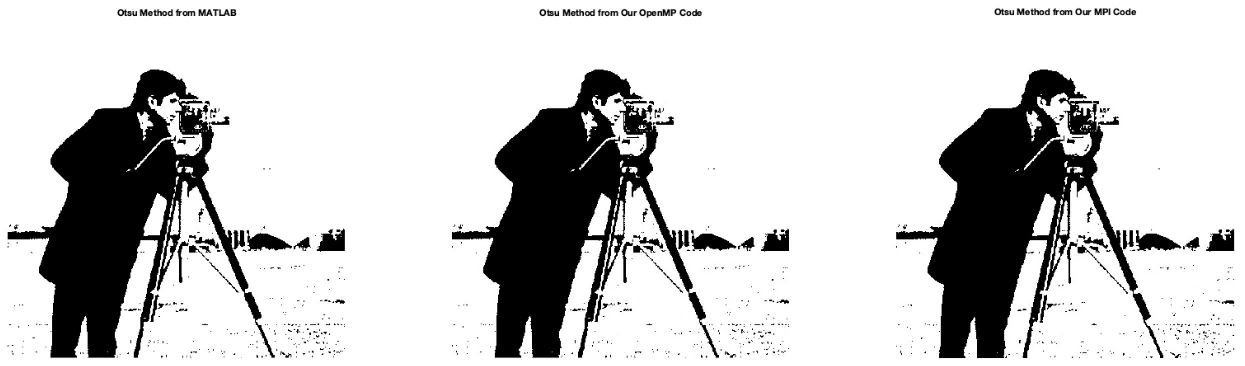

Otsu Method from MATLAB
Otsu Method from Our OpenMP Code
Otsu Method from Our MPI Code

Figure 7: Segmented 512x512 images with Otsu using MATLAB (*left*), OpenMPI (*center*) and MPI (*right*)

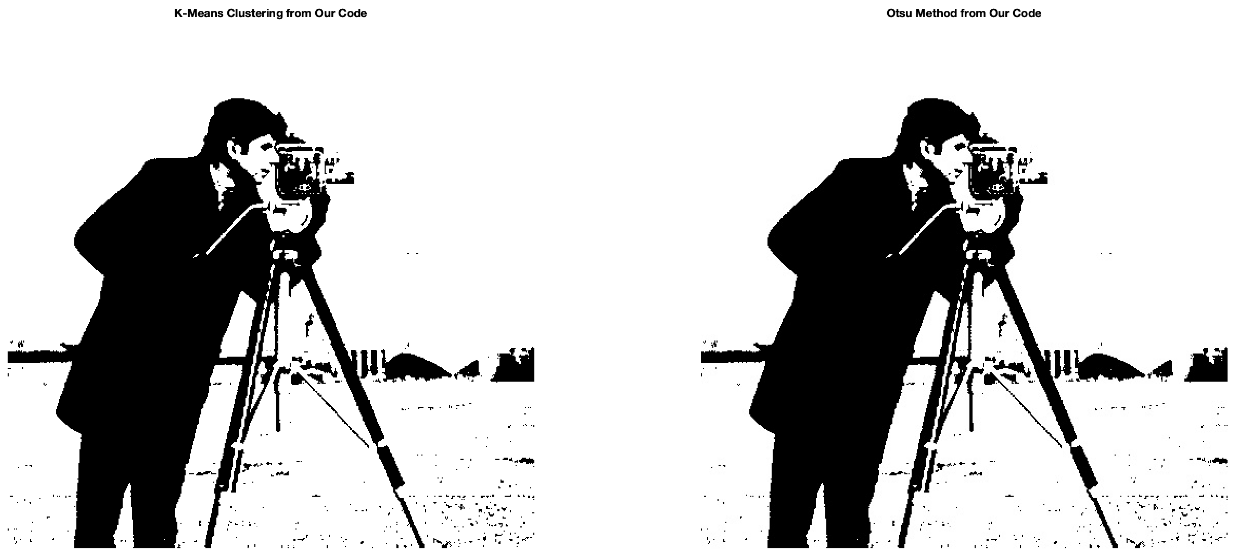


Figure 8: Segmented 512x512 images from K-Means Clustering vs. Otsu's Method

4.2 Performance Analysis

In our implementation setup, we selectively measure the execution time of the parallel segment of the algorithm. Alternative plots for execution time can be found in appendix c.

Table 1: Structural Similarity (SSIM) Index for K-Means and Otsu with a 512x512 image using MATLAB, OpenMP and MPI

Interface	Algorithm	SSIM MATLAB
OpenMP	K-Means	0.9982
	Otsu	1.0000
MPI	K-Means	0.9982
	Otsu	1.0000

Table 2: Structural Similarity (SSIM) Index for K-Means using OpenMP vs. MPI and for Otsu using OpenMP vs. MPI with a 512x512 image

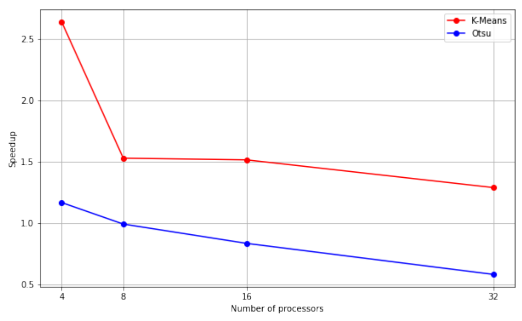
Interface	SSIM
	K-Means vs. Otsu
OpenMP	0.9877
MPI	0.9877

Table 3: Execution Times for K Means and Otsu Algorithms in Serial

	Image Size	Time (s)	
		K Means	Otsu
Serial	128x128	0.002191020	0.000761020
	256x256	0.005195690	0.001608840
	512x512	0.029598400	0.004504510

Table 4: Execution Times for K Means and Otsu with a 128x128 image using OpenMP and MPI

Interface	Processors	Time (s)	
		K Means	Otsu
OpenMP	p = 4	0.000829394	0.000651022
	p = 8	0.001431740	0.000766023
	p = 16	0.001444590	0.000910017
	p = 32	0.001697240	0.001303820
MPI	p = 4	0.000462471	0.000685554
	p = 8	0.000514324	0.000729291
	p = 16	0.000544384	0.000861183
	p = 32	0.002738080	0.005067740



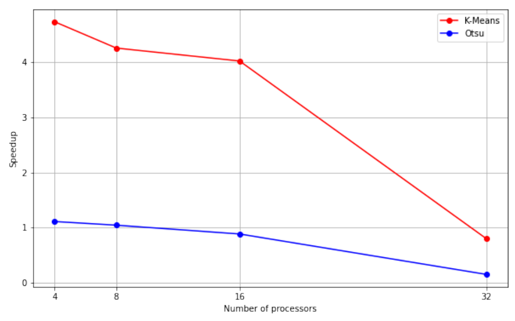


Figure 9: Speedup of OpenMP (*left*) and MPI (*right*) with a 128x128 image

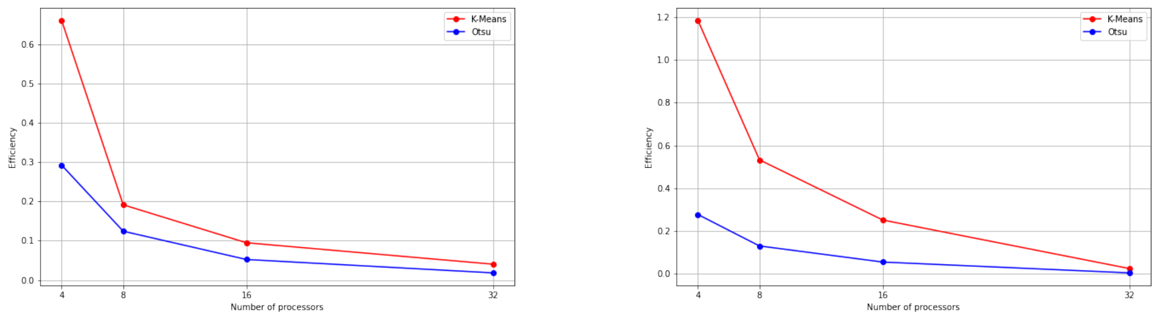


Figure 10: Efficiency of OpenMP (*left*) and MPI (*right*) with a 128x128 image

Table 5: Execution Times for K Means and Otsu with a 256x256 image using OpenMP and MPI

Interface	Processors	Time (s)	
		K Means	Otsu
OpenMP	p = 4	0.001493470	0.00134610
	p = 8	0.001780670	0.001395050
	p = 16	0.001732730	0.001319450
	p = 32	0.001546330	0.001337910
MPI	p = 4	0.000710182	0.000546008
	p = 8	0.000522460	0.000544481
	p = 16	0.000472720	0.000577407
	p = 32	0.002059920	0.001707830

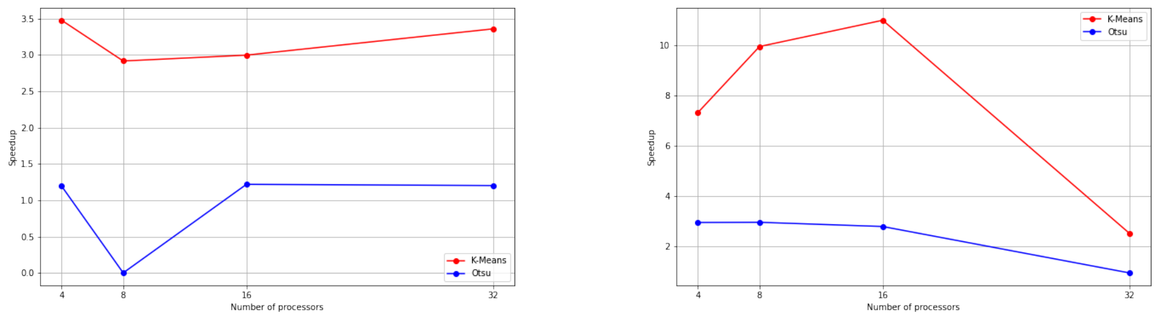
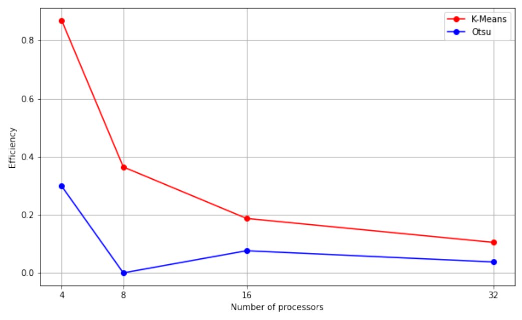


Figure 11: Speedup of OpenMP (*left*) and MPI (*right*)) with a 256x256 image



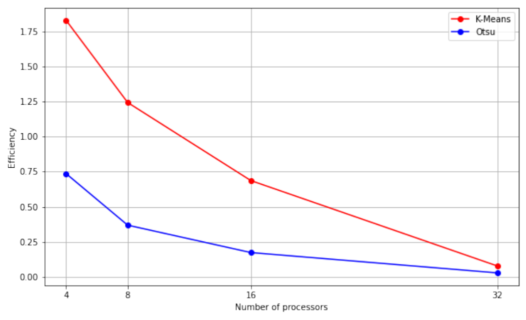
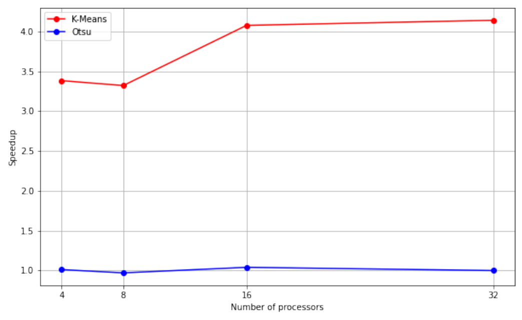


Figure 12: Efficiency of OpenMP (*left*) and MPI (*right*) with a 256x256 image

Table 6: Execution Times for K Means and Otsu with a 512x512 image using OpenMP and MPI

Interface	Processors	Time (s)	
		K Means	Otsu
OpenMP	p = 4	0.008751030	0.004449880
	p = 8	0.008913500	0.004638140
	p = 16	0.007262410	0.004326970
	p = 32	0.007150470	0.004502110
MPI	p = 4	0.003495210	0.000491196
	p = 8	0.002049230	0.000569665
	p = 16	0.001337140	0.000548115
	p = 32	0.003850330	0.001696960



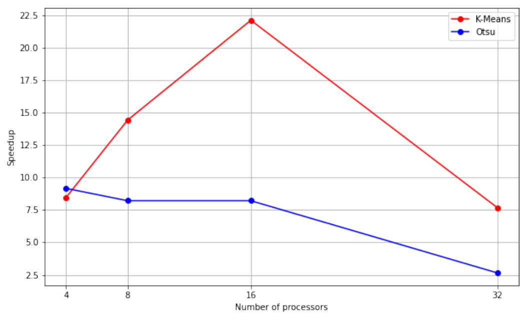
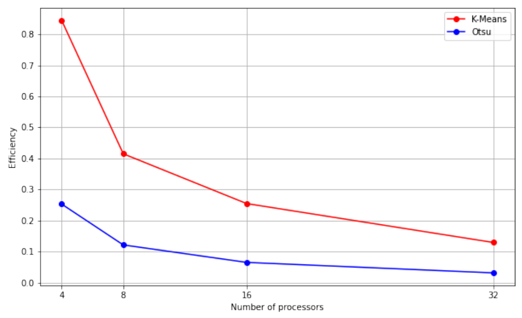


Figure 13: Speedup of OpenMP (*left*) and MPI (*right*) with a 512x512 image



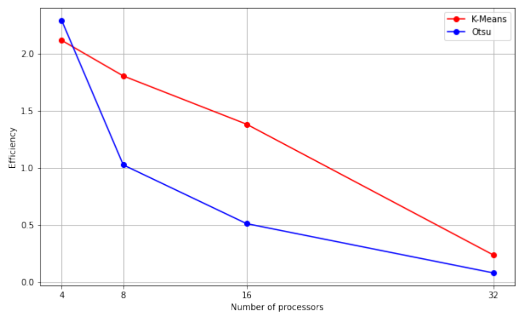


Figure 14: Efficiency of OpenMP (*left*) and MPI (*right*) with a 512x512 image

5 Discussion

5.1 Findings

For OpenMP, both K-Means and Otsu showed consistency in timing as the number of processors increased. However, for MPI, K-Means, we observed the program slowing down with a image size of 128x128 and 256x256 whereas Otsu was consistent.

With a small image size 128x128, a small number of processors $p = 4$ is recommended to minimize the communication overheads. As the image size increases 2 times larger to a 256x256 image, it is still acceptable to use 4 processors. With a 4 times larger image than a 128x128 image, 16 processors is more favorable in timing for 512x512 image. In particular, Otsu implemented with MPI is faster than K-Means implemented with OpenMP with a image size of 128x128 and it is apparently much faster with a larger image size.

When we look at the speed-up and efficiency of each algorithm corresponding to three different image sizes, we observe the speedup for K-Means was higher than Otsu, showing it is much faster than its serial code. However, we can see the speedup for MPI K-Means drastically drop when the number of processors reaches 32, which we observed in the timing. The reason why the speedup decreases as $p = 32$ is that we cannot request two nodes on Tuckoo cluster to run the code at the same time in the command line. Both algorithms implementing OpenMP and MPI showed a decrease in efficiency as the number of processors increased, following an negative exponential curve shape. Despite Otsu being more consistent, K-Means often fared slightly better than Otsu in both speedup and efficiency. However, K-Means begins to converge towards Otsu for 32 processors, so there is potential for Otsu to overcome K-Means as processors increase.

We can interpret this to mean that K-Means and Otsu fare better than their serial counterparts, however that speed and efficiency decline as we increase the number of processors. There can be many reasons as to why an increase in processors leads to a decrease in quality. One reason may be that there is an optimal processor to image size ratio that needs to be met for the algorithm to be fast and efficient. Another reason could be that the higher number of processors means there needs to be higher communication between processors, ultimately slowing the algorithm down. Perhaps with more images of larger size, we would be able to tell the relationship between processors and speed/efficiency more clearly.

It is also important to note that the size of the image was always divisible by the number of processors, meaning the distribution of tasks was always of equal size. It is possible with a non-divisible processor count, the results could vary.

5.2 Difficulties

5.2.1 K-Means

The K-Means algorithm requires multiple loops to complete. In serial and OpenMP, this was not an issue because memory was shared. For OpenMP, there was also the ability to have parallel and serial sections. This made communication between threads very simple. This meant that the OpenMP code was not very different from the serial code. For MPI, this was not the case. The code for MPI had to be completely reworked to include allow all processors to perform the loop based on broadcasts from the main processor. The division of tasks was also very different. With serial and OpenMP, each processor can view and alter the original and final image without interference. For MPI, the main thread had to scatter the data of the image, all processors perform segmentation, then all data is gathered to produce the final image. This, coupled with the additional code required for communication between processors, lead the MPI code to be much longer and more time consuming than the other codes.

5.2.2 Otsu's Method

The Otsu algorithm requires multiple steps after it computes the histogram and probability of each intensity value. In particular, it requires a loop that goes all possible gray scale level values with a initial threshold to separate an image pixels into foreground class and background class and then search for optimal threshold which holds the maximum between class variance. It is not hard to implement this algorithm in serial and OpenMP. In OpenMP, every thread is responsible in maintaining their private copy of the between class variances and the probability data is shared among the threads. OpenMP automatically distribute the computation across multiple threads, where each thread computes the cumulative means, cumulative weights, total mean and keep track of maximum between class variance value. However, implementing Otsu algorithm with MPI is more challenging since each thread has its own private memory and the probability data sharing among these threads require explicit communication. If the probability data is evenly distributed among the threads, the optimal threshold would be changed with different number of threads. The only way to avoid having different optimal threshold values is to send a copy of probability data to all processes. Each process can perform their calculations for the cumulative means, cumulative weights, total mean and find the maximum between class variance. At the end, all maximum between class variances obtained from each process are brought together to find one global maximum between class variance and its corresponding threshold.

6 Conclusions

From our findings, Otsu and K-Means both have their benefits and shortcomings. Otsu proved to be more accurate compared to the MATLAB results, however fell short of K-Means in speedup and efficiency. K-Means, however, began to decrease in speedup and efficiency as the number of processors increased, whereas Otsu was consistent in quality.

Based on this, it may be safe to say Otsu is more reliable. The limitations of Otsu is that the algorithm can only segment an image into two regions. K-Means is capable of segmenting an image into as many regions as there are color values. Depending on specific requirements, one algorithm may prove more suitable than the other.

In terms of parallel languages, OpenMP was both user-friendly and faster to work with than MPI. The ability to have parallel and serial sections has decidedly made OpenMP more favorable. However, MPI was almost always faster than OpenMP despite the complicated code. Our project was not dependent on time, however, there may be cases where the speed of MPI would make it preferable to OpenMP.

7 Division of Work

Each of us implemented one image segmentation method in both OpenMP and MPI. We all worked together to debug, test, and analyze the results of the project. We met regularly to collaborate and helped each other. In no specific order, we list our main contributions.

Luna Huynh - Otsu's Method, MATLAB images

Daisy Ulloa - K-Means clustering, test images

Group effort: Slides, Report, Result analysis (timing, speedup, efficiency)

References

[1] Kshitiz Bhatt. "A Comparative Analysis of Otsu Thresholding and K-means Algorithm for Image Segmentation". PhD thesis. Department of Computer Science and Information Technology, 2019.

[2] Ajay Padoor Chandramohan. *Parallel K-means Clustering*. https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Chandramohan-Fall-2012-CSE633.pdf. 2012.

[3] Milan Jaroš et al. "Implementation of K-means segmentation algorithm on Intel Xeon Phi and GPU: Application in medical imaging". In: *Advances in Engineering Software* 103 (2017), pp. 21–28.

[4] Dongju Liu and Jian Yu. "Otsu method and K-means". In: *2009 Ninth International conference on hybrid intelligent systems*. Vol. 1. IEEE. 2009, pp. 344–349.

Appendix

A Code Access and Compilation

All the code can be accessed through Daisy's tuckoo folder. The files can be found under the following two folders:

```
home/605/ulloa/project/kMeans
home/605/ulloa/project/otsu
```

Each folder will have images to run the code, as well as a subfolder /miscFiles for bash scripts and other previous codes.

The c files will be of the form: <method><API method>.c where <method> is either kMeans or otsu and API method is either openMP, MPI, or Serial.

To compile and run serial code, enter the following into the command line:

```
$ g++ -Wall -g -o <filename> <filename>.c ‘pkg-config --cflags --libs opencv‘
$ ./<filename>
```

For compiling and running OpenMP files, enter the following in command line:

```
$ g++ -Wall -g -o <filename> <filename>.c -fopenmp ‘pkg-config --cflags --libs opencv‘
$ ./<filename> <number of threads>
```

For compiling and running MPI files, enter the following in command line:

```
$ mpicxx -Wall -g -o <filename> <filename>.c ‘pkg-config --cflags --libs opencv‘
$ mpirun --oversubscribe -n <number of threads> ./<filename>
```

Once compiled and run, the segmented images will save to a subfolder titled <method>/images. Image files will be titled as: camera<method><size>_<API method>.jpg

B MATLAB Code

The following MATLAB code is used to segment a camera man image using K-Means clustering method with $k = 2$. The left image of Figure 6 is obtained from this code.

```matlab
% Read an image
img = imread('camera512.jpg');

% Convert to grayscale if it's a color image
if size(img, 3) == 3
    img = rgb2gray(img);
end
% Use imsegkmeans to segment the image into 2 clusters
[L, Centers] = imsegkmeans(img, 2);

% Determine which cluster corresponds to the darker pixels (assuming you want dark as
    background)
if Centers(1) < Centers(2)
    darkCluster = 1;
else
    darkCluster = 2;
end

% Create a binary image where the dark cluster is black (0) and the light cluster is white
    (255)
binaryImage = uint8(L == darkCluster) * 0 + uint8(L ~= darkCluster) * 255;

```

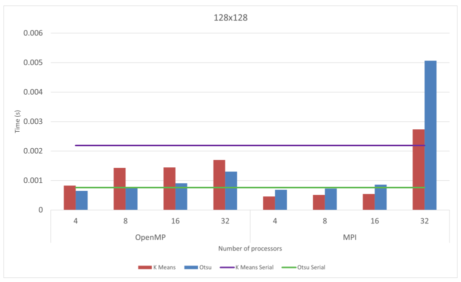
```
IK_op=imread('camera512kMeans_OP.jpg');
IK_mpi=imread('camera512kMeans_MPI.jpg');
% Display the segmented images
figure(1);
subplot(1,3,1);
imshow(binaryImage, []);
title('K-Means Clustering from MATLAB');

subplot(1,3,2);
imshow(IK_op);
title('K-Means Clustering from Our OpenMP Code');

subplot(1,3,3);
imshow(IK_mpi);
title('K-Means Clustering from Our MPI Code');

IO_mpi = imread('cameraOtsu512_MPI.jpg');
IO_op = imread('cameraOtsu512_OpenMP.jpg');
figure(2);
subplot(1,2,1);
imshow(IK_mpi, []);
title('K-Means Clustering from Our MPI Code');

subplot(1,2,2);
imshow(IO_mpi);
title('Otsu Method from Our MPI Code');

ssim_Kmeans_MATLAB_OpenMP=ssim(binaryImage, IK_op); %0.9982
% Calulate SSIM between K-Means segmented images obtained from MATLAB and from our MPI
ssim_Kmeans_MATLAB_MPI=ssim(binaryImage, IK_mpi); %0.9982
ssim_Kmeans_OpenMP_MPI=ssim(IK_op, IK_mpi); %1.0000
ssim_Kmeans_Otsu_MPI=ssim(IK_mpi, IO_mpi); %0.9877
ssim_Kmeans_Otsu_OpenMP=ssim(IK_op, IO_op); %0.9877
```

The following MATLAB code is used to segment a camera man image using Otsu's Method. The left image of Figure 7 is obtained from this code.

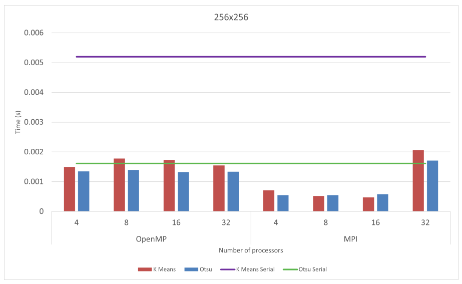
```
I = imread('camera512.jpg');
I = rgb2gray(I);
IO_op = imread('cameraOtsu512_OpenMP.jpg');
IO_mpi = imread('cameraOtsu512_MPI.jpg');
% Ensure I is two-dimensional
assert(ismatrix(I), 'The input image I is not two-dimensional.');

% Calculate the optimal threshold using Otsu's method
threshold = graythresh(I);

% Apply the threshold to produce a binary image
binaryImg = imbinarize(I, threshold);

% Check the dimensions of binaryImg
disp(size(binaryImg));

% Display the original and binary images
figure;
subplot(1,3,1);
imshow(binaryImg);
title('Otsu Method from MATLAB');
subplot(1,3,2);
imshow(IO_op);
title('Otsu Method from Our OpenMP Code');
subplot(1,3,3);
imshow(IO_mpi);
title('Otsu Method from Our MPI Code');

% Invert the image
bI = uint8((binaryImg));
% Convert any 0 to 255 (after converting initial 255s to 0)
IO_op = IO_op/255;
```

```
IO_mpi = IO_mpi/255;

% Calculate SSIM between the MATLAB and MPI
ssimg1=ssim(bI, IO_mpi); %1.0000
% Calculate SSIM between the MATLAB and OpenMP
ssimg2=ssim(bI, IO_op); %1.0000
% Calculate SSIM between the OpenMP and MPI
ssimg3=ssim(IO_op,IO_mpi); %1.0000
```

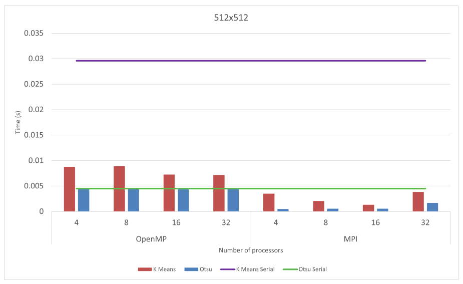
C More Plots



(a) 128x128



(b) 256x256



(c) 512x512

Figure 15: Execution time of methods with serial time