High-Performance Computing

OpenMP k-Means

Anand Kamble
Department of Scientific Computing
Florida State University

1 Introduction

In this project, we explore the application of the k-means algorithm for image segmentation, leveraging OpenMP for parallelization to enhance performance. Image segmentation involves partitioning an image into multiple segments or regions based on certain characteristics. K-means clustering is a popular unsupervised learning algorithm for clustering tasks, including image segmentation.

2 Implementation

In this project, four main parallel loops are employed to execute the image segmentation process efficiently. These loops are contained within the parallelized region of the program. Before looking into the details of the parallelized loops, it's crucial to understand the non-parallelized part, particularly the initialization phase, which sets the foundation for the segmentation algorithm.

3 Non Parallerlized section

This section includes reading the image from the file and storing it in an array, along with its height and width into an unsigned integer.

```
dataBuf = JpegFile::JpegFileToRGB("Images/test.jpg", &width, &height);
```

In this section, we are also setting the parameters of our programs such as the number of clusters and the number of iterations for the algorithm. Also, we are creating the Groups i.e. the clusters for the K-means algorithm, and setting their initial generators by simply setting the generator pixel to be every $(\frac{N}{k})$th pixel, where N is the total number of pixels and k is the number of clusters.

```
Group *groups = new Group[k]; // Array to store groups

    for (int i = 0; i < k; i++)
    {
        groups[i].setGenerator(dataBuf[(((N * 3) / k) * i)],
                               dataBuf[(((N * 3) / k) * i) + 1],
                               dataBuf[(((N * 3) / k) * i) + 2]);

        groups[i].setpixels((int *)malloc(sizeof(int) * (N * 3)));
    }
```

Here we are also allocating memory for the pixels that will be in that group. We are using the size as $N \times 3$ since we are storing the RGB values and there can be an edge case where all the pixels lie in the same group.
We are also setting the number of threads, and also making sure that we don't exceed the maximum number of threads available.

```
    int NUM_THREADS = 32;
    NUM_THREADS = min(NUM_THREADS, omp_get_max_threads());
    omp_set_num_threads(NUM_THREADS);
```

At the end of the program, there is also a non-parallelized part that handles writing the segmented image to the disk and deallocates the memory used by the image and the groups.

```
    JpegFile::RGBToJpegFile("test_seg.jpg", dataBuf, width, height,
        true, 100);
    delete dataBuf;
    delete[] groups;
```

4 Parallerlized section

The whole region that needs to be parallelized is enclosed in brackets like this:

```
#pragma omp parallel
    {
    ... Parallel region
    }
```

4.1 Initializing the group

During every iteration, we have to remove the pixels from every group keeping the generators and the average color of that group. For this we are using `clearPixels` function from the `Group` class.

```
#pragma omp for schedule(static)
        for (int i = 0; i < k; i++)
        {
            groups[i].clearPixels();
        }
```

This process is quite fast but in case the number of clusters is very large, it will benefit from the parallelization.

4.2 Grouping the pixels

In this part, we are grouping the pixels by looping over all of them finding which generator is closest to them, and then adding that pixel to the respective group.

```
#pragma omp for schedule(static)
        for (int i = 0; i < N; i++)
        {
            DIST minimumDistance = INFINITY;
            Group *minimumDistanceGroup = nullptr;
            for (int j = 0; j < k; j++)
            {
                DIST distance = distanceBetweenPexels(dataBuf, i,
                    groups[j].generator);
                if (distance < minimumDistance) {
                    minimumDistance = distance;
                    minimumDistanceGroup = &groups[j];
                }
            }
            if (minimumDistanceGroup != nullptr)
            {
                minimumDistanceGroup->setPixelCount(
                    minimumDistanceGroup->pixelCount + 1
                );
                minimumDistanceGroup->addPixel(i);
            }
        }
```

Using the static schedule in this part, since we have to loop over all the pixels, and for each pixel loop over all the groups. This process for each pixel should take a similar time and using the dynamic schedule for a large number of N might slow down the program.

4.3 Get new generators

Once we have all the pixels, we find the average color of the pixels by looping through all the pixels in each cluster. Here we are using '*dynamic*' scheduling since each cluster might contain a different number of pixels.

```
#pragma omp for schedule(dynamic)
    for (int i = 0; i < k; i++)
    {
        int numOfPixels = groups[i].getPixelCount();
        long long int averageR = 0, averageG = 0, averageB = 0;
        for (int j = 0; j < numOfPixels; j++) {
            BYTE *pTest = dataBuf + (groups[i].getpixels()[j] * 3);
            averageR += (*pTest);        // Red
            averageG += (*(pTest + 1)); // Green
            averageB += (*(pTest + 2)); // Blue
        }
        if (numOfPixels > 0) {
            averageR /= numOfPixels;
            averageG /= numOfPixels;
            averageB /= numOfPixels;
            groups[i].setGenerator(averageR, averageG, averageB);
            groups[i].average[0] = averageR; // Red
            groups[i].average[1] = averageG; // Green
            groups[i].average[2] = averageB; // Blue
        }
    }
```

4.4 Finishing

Once we have completed all the iterations of the k-means algorithm, we update the image buffer by replacing the color of each pixel with the average of the group that it belongs to.

```
    #pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < k; i++)
    {
        auto groupPixels = groups[i].getpixels();
        auto pixelCount = groups[i].getPixelCount();
        auto average = groups[i].average;
        for (int j = 0; j < pixelCount; j++)
        {
            auto pTest = dataBuf + (groupPixels[j] * 3);
            // Coloring the groups
            *pTest = average[0];
            *(pTest + 1) = average[1];
            *(pTest + 2) = average[2];
        }
    }
```

5 Performance Improvements

This program has been benchmarked using the following numbers of threads: 1, 2, 4, 8, 10, and 12. With $k = 5$ and a image of size 640×960.

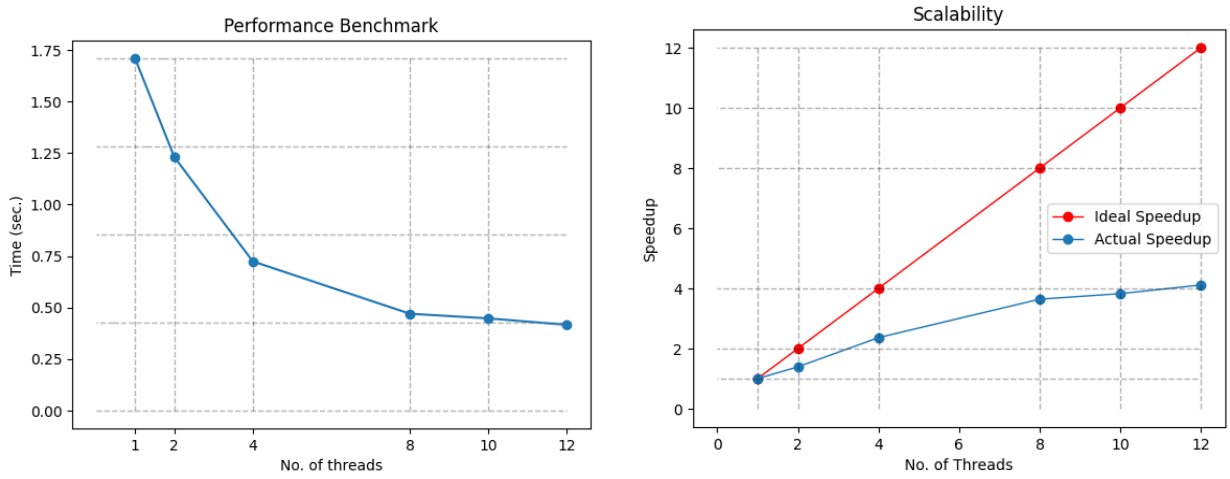


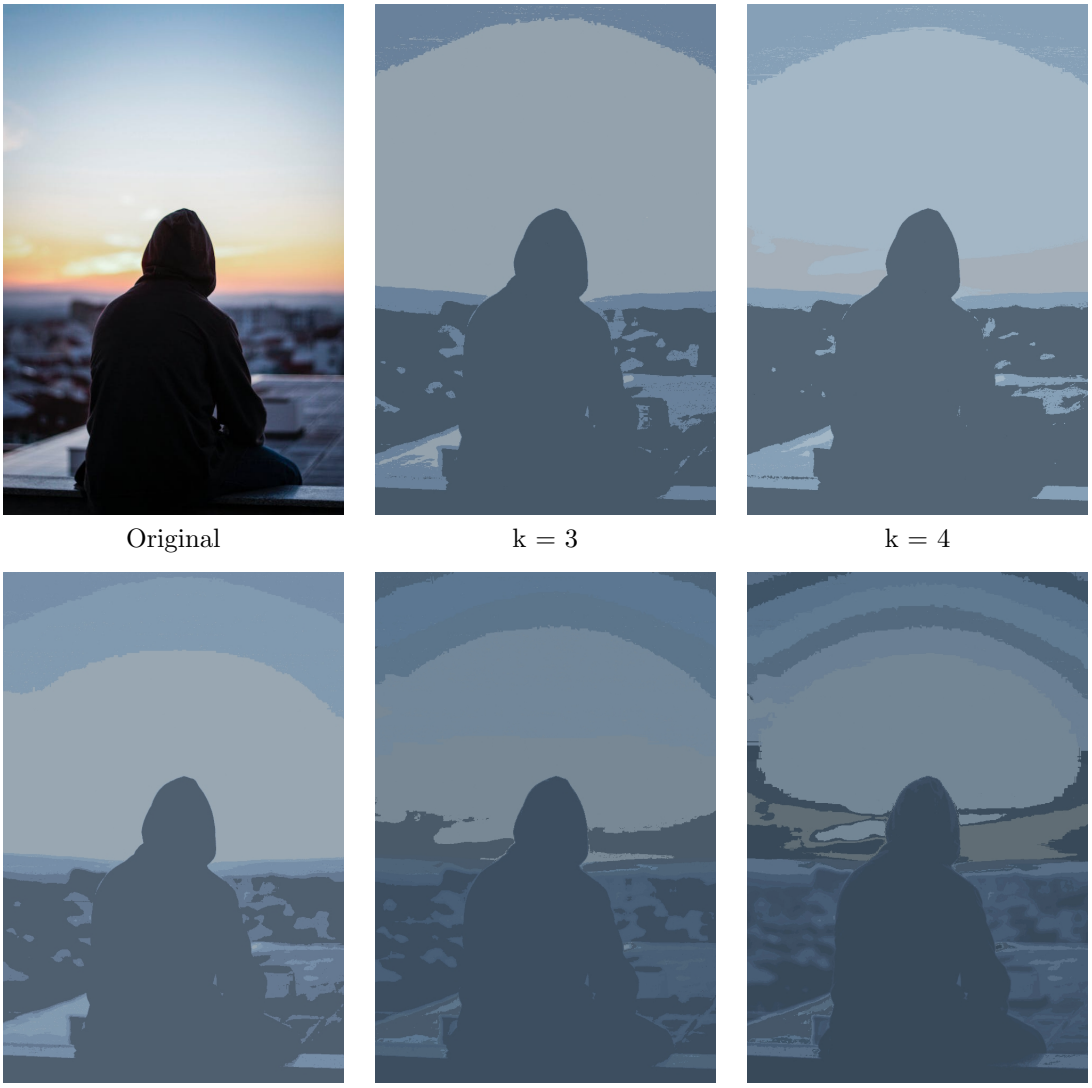
Figure 1: Timing and Scaling of the program

6 Errors and Debugging

During the development, the major issue faced was getting horizontal lines in the output image, this was due to updating the colors of the pixels at last rather than at every iteration.

Also, there were many nested parallel loops in the initial versions of the code, which were not improving performance.

7 Output



Original | k = 3 | k = 4

k = 5 | k = 10 | k = 20

Figure 2: Output Images comparison