Image Compression Using K-Means Clustering with OpenMP

Computer Architecture and Organization – DA 3

Sanchit Mukherjee – 23BCE5126

Objective of the Project:

The primary objective of this project is to design and implement a K-Means based image compression system in C, utilizing OpenMP to explore the performance benefits of parallel computing. The system aims to:

- Compress images by reducing their colour space using K-Means clustering.
- Compare serial and parallel implementations with varying thread counts to analyse execution time.
- Investigate how thread parallelism affects scalability and performance across different k cluster values.
- Evaluate the trade-off between image quality and compression ratio based on the chosen k.
- Visualize execution time trends using R to identify optimal configurations.

By the end of this project, the goal is to highlight how parallelization improves performance in image compression tasks, and how tuning parameters like cluster size (k) can help balance processing speed, file size, and visual quality.

CODE:

K-means Compression

kmeans_compressor.c > ...

```c
#define STB_IMAGE_IMPLEMENTATION
#define STB_IMAGE_WRITE_IMPLEMENTATION

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include "stb_image.h"
#include "stb_image_write.h"

#define INPUT_FILE "input.png"
#define OUTPUT_TEMPLATE "output_%d.png"

int main(int argc, char *argv[])
{
    if (argc < 2)
    {
        printf("Usage: %s <k>\n", argv[0]);
        return 1;
    }

    int k = atoi(argv[1]);
    int max_threads = omp_get_num_procs();

    int width, height, channels;
    unsigned char *image = stbi_load(INPUT_FILE, &width, &height, &channels, 3);
    if (!image)
    {
        fprintf(stderr, "Failed to load %s\n", INPUT_FILE);
        return 1;
    }

    int img_size = width * height;
    FILE *log = fopen("performance_log.csv", "a");
    if (log)
        fprintf(log, "k,width,height,num_threads,time_seconds\n");

    for (int num_threads = 1; num_threads <= max_threads; num_threads++)
    {
        omp_set_num_threads(num_threads);

        unsigned char *output = (unsigned char *)malloc(img_size * 3);
        float *centroids = (float *)malloc(k * 3 * sizeof(float));
        int *labels = (int *)malloc(img_size * sizeof(int));

        // Init centroids randomly
        for (int i = 0; i < k; i++)
        {
            int idx = rand() % img_size;
            centroids[i * 3 + 0] = image[idx * 3 + 0];
            centroids[i * 3 + 1] = image[idx * 3 + 1];
```

```c
            centroids[i * 3 + 2] = image[idx * 3 + 2];
        }

        double start_time = omp_get_wtime();

        for (int iter = 0; iter < 10; iter++)
        {
#pragma omp parallel for
            for (int i = 0; i < img_size; i++)
            {
                float min_dist = 1e9;
                int best = 0;
                for (int j = 0; j < k; j++)
                {
                    float dr = image[i * 3 + 0] - centroids[j * 3 + 0];
                    float dg = image[i * 3 + 1] - centroids[j * 3 + 1];
                    float db = image[i * 3 + 2] - centroids[j * 3 + 2];
                    float dist = dr * dr + dg * dg + db * db;
                    if (dist < min_dist)
                    {
                        min_dist = dist;
                        best = j;
                    }
                }
                labels[i] = best;
            }

            float *new_centroids = (float *)calloc(k * 3, sizeof(float));
            int *counts = (int *)calloc(k, sizeof(int));

#pragma omp parallel for
            for (int i = 0; i < img_size; i++)
            {
                int j = labels[i];
#pragma omp atomic
                new_centroids[j * 3 + 0] += image[i * 3 + 0];
#pragma omp atomic
                new_centroids[j * 3 + 1] += image[i * 3 + 1];
#pragma omp atomic
                new_centroids[j * 3 + 2] += image[i * 3 + 2];
#pragma omp atomic
                counts[j]++;
            }

            for (int j = 0; j < k; j++)
            {
                if (counts[j] > 0)
                {
```

```c
                centroids[j * 3 + 0] = new_centroids[j * 3 + 0] / counts[j];
                centroids[j * 3 + 1] = new_centroids[j * 3 + 1] / counts[j];
                centroids[j * 3 + 2] = new_centroids[j * 3 + 2] / counts[j];
            }
        }

        free(new_centroids);
        free(counts);
    }

    double end_time = omp_get_wtime();
    double elapsed = end_time - start_time;

// Generate image
#pragma omp parallel for
    for (int i = 0; i < img_size; i++)
    {
        int j = labels[i];
        output[i * 3 + 0] = (unsigned char)centroids[j * 3 + 0];
        output[i * 3 + 1] = (unsigned char)centroids[j * 3 + 1];
        output[i * 3 + 2] = (unsigned char)centroids[j * 3 + 2];
    }

    char output_filename[64];
    snprintf(output_filename, sizeof(output_filename), OUTPUT_TEMPLATE, num_threads);
    stbi_write_png(output_filename, width, height, 3, output, width * 3);

    printf("[Threads: %2d] Time = %.4f sec - Saved: %s\n", num_threads, elapsed, output_filename);

    if (log)
    {
        fprintf(log, "%d,%d,%d,%d,%.6f\n", k, width, height, num_threads, elapsed);
    }

    free(output);
    free(centroids);
    free(labels);
  }

  if (log)
      fclose(log);
  stbi_image_free(image);
  return 0;
}

// gcc -fopenmp kmeans_compressor.c -o kcompress.exe -lm
```

Visualization

```r
threads <- 1:16

time_k16 <- c(1.993000,1.567000,1.408000,1.244000,1.310000,1.213000,1.093000,1.244000,1.397000,
              1.327000,1.451000,1.313000,1.500000,1.387000,1.286000,1.409000)
time_k32 <- c(3.535000,2.305000,1.868000,1.758000,1.600000,1.538000,1.490000,1.280000,1.430000,
              1.268000,1.489000,1.471000,1.297000,1.394000,1.398000,1.406000)
time_k64 <- c(6.632000,4.119000,3.129000,2.705000,2.394000,2.344000,2.016000,2.054000,1.809000,
              1.923000,1.939000,1.781000,1.615000,1.804000,1.818000,1.720000)
time_k128 <- c(12.909000,7.961000,6.046000,4.659000,4.157000,3.790000,3.547000,3.298000,3.165000,
               2.960000,2.778000,2.684000,2.611000,2.621000,2.440000,2.578000)
time_k256 <- c(25.208000,15.431000,11.785000,9.950000,8.358000,7.646000,6.771000,6.017000,5.732000,
               5.189000,4.997000,4.807000,4.633000,4.400000,4.280000,4.422000)
time_k512 <- c(50.475000,36.716000,24.801000,20.602000,16.924000,15.128000,12.999000,12.052000,10.929000,10.

plot(threads, time_k16, type = "o", col = "red", pch = 16, ylim = c(0.5, 20),
     xlab = "Number of Threads", ylab = "Time (seconds)",
     main = "K-Means Compression Time (Multiple k Values)")
lines(threads, time_k32, type = "o", col = "blue", pch = 17)
lines(threads, time_k64, type = "o", col = "darkgreen", pch = 18)
lines(threads, time_k128, type = "o", col = "purple", pch = 15)
lines(threads, time_k256, type = "o", col = "orange", pch = 3)
lines(threads, time_k512, type = "o", col = "brown", pch = 4)
legend("topright", legend = c("k = 16", "k = 32", "k = 64", "k = 128", "k = 256", "k = 512"),
       col = c("red", "blue", "darkgreen", "purple", "orange", "brown"),
       pch = c(16, 17, 18, 15, 3, 4),
       title = "Cluster Count (k)")
```

OUTPUT:

Input Image

Figure 1: Input Image (Size - 3.4 MB)

Output Images

Figure 2: Centroids = 16 (size - 1.2 MB)

Figure 3: Centroids = 32 (size - 1.5 MB)

Figure 4: Centroids = 64 (size - 1.8 MB)

Figure 5: Centroids = 128 (size - 2.1 MB)

Figure 6: Centroids = 256 (size - 2.3 MB)

Figure 7: Centroids = 512 (size - 2.6 MB)

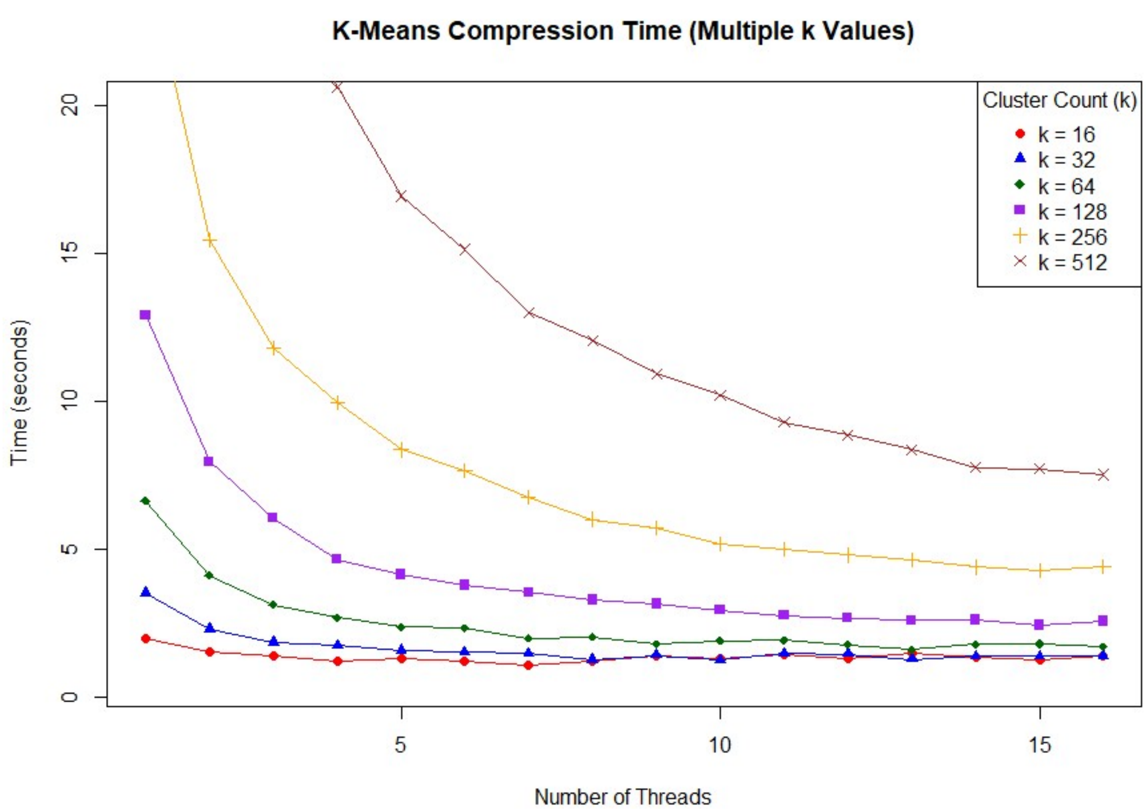
Data logged

K	width	height	No. of threads	Time Taken (s)
16	1792	1024	1	1.993
16	1792	1024	2	1.567
16	1792	1024	3	1.408
16	1792	1024	4	1.244
16	1792	1024	5	1.31
16	1792	1024	6	1.213
16	1792	1024	7	1.093
16	1792	1024	8	1.244
16	1792	1024	9	1.397
16	1792	1024	10	1.327
16	1792	1024	11	1.451
16	1792	1024	12	1.313
16	1792	1024	13	1.5
16	1792	1024	14	1.387
16	1792	1024	15	1.286
16	1792	1024	16	1.409
32	1792	1024	1	3.535
32	1792	1024	2	2.305
32	1792	1024	3	1.868
32	1792	1024	4	1.758
32	1792	1024	5	1.6
32	1792	1024	6	1.538
32	1792	1024	7	1.49
32	1792	1024	8	1.28
32	1792	1024	9	1.43
32	1792	1024	10	1.268
32	1792	1024	11	1.489
32	1792	1024	12	1.471

32	1792	1024	13	1.297
32	1792	1024	14	1.394
32	1792	1024	15	1.398
32	1792	1024	16	1.406
64	1792	1024	1	6.632
64	1792	1024	2	4.119
64	1792	1024	3	3.129
64	1792	1024	4	2.705
64	1792	1024	5	2.394
64	1792	1024	6	2.344
64	1792	1024	7	2.016
64	1792	1024	8	2.054
64	1792	1024	9	1.809
64	1792	1024	10	1.923
64	1792	1024	11	1.939
64	1792	1024	12	1.781
64	1792	1024	13	1.615
64	1792	1024	14	1.804
64	1792	1024	15	1.818
64	1792	1024	16	1.72
128	1792	1024	1	12.909
128	1792	1024	2	7.961
128	1792	1024	3	6.046
128	1792	1024	4	4.659
128	1792	1024	5	4.157
128	1792	1024	6	3.79
128	1792	1024	7	3.547
128	1792	1024	8	3.298
128	1792	1024	9	3.165
128	1792	1024	10	2.96
128	1792	1024	11	2.778
128	1792	1024	12	2.684
128	1792	1024	13	2.611
128	1792	1024	14	2.621
128	1792	1024	15	2.44
128	1792	1024	16	2.578
256	1792	1024	1	25.208
256	1792	1024	2	15.431
256	1792	1024	3	11.785
256	1792	1024	4	9.95
256	1792	1024	5	8.358
256	1792	1024	6	7.646
256	1792	1024	7	6.771
256	1792	1024	8	6.017
256	1792	1024	9	5.732
256	1792	1024	10	5.189

256	1792	1024	11	4.997
256	1792	1024	12	4.807
256	1792	1024	13	4.633
256	1792	1024	14	4.4
256	1792	1024	15	4.28
256	1792	1024	16	4.422
512	1792	1024	1	50.475
512	1792	1024	2	36.716
512	1792	1024	3	24.801
512	1792	1024	4	20.602
512	1792	1024	5	16.924
512	1792	1024	6	15.128
512	1792	1024	7	12.999
512	1792	1024	8	12.052
512	1792	1024	9	10.929
512	1792	1024	10	10.209
512	1792	1024	11	9.298
512	1792	1024	12	8.859
512	1792	1024	13	8.374
512	1792	1024	14	7.772
512	1792	1024	15	7.715
512	1792	1024	16	7.528

Time Taken Vs Number of Threads



Findings:

1) Parallelization Significantly Reduces Execution Time

- As the number of threads increases from 1 to 16, execution time consistently decreases for all tested k values.
- The highest performance gain is observed at lower thread counts (2–8), indicating efficient parallelization using OpenMP.

2) Diminishing Returns at Higher Thread Counts

- Beyond 8 to 12 threads, the performance curve flattens, especially for lower k values.
- This suggests overhead from thread management and limited parallel workload at smaller cluster sizes.

3) Larger k Values Benefit More from Parallelization

- Higher k values (e.g., 256 and 512) yield more noticeable time reduction across increasing threads, indicating better scalability due to higher computational demands.

4) Compression Ratio vs. Quality Trade-off

- The original image was 3.4 MB. After compression:
  - k = 16 produced a file of 1.2 MB (≈65% size reduction).
  - k = 512 produced a file of 2.6 MB (≈24% reduction).
- This shows that lower k results in higher compression but with more aggressive colour quantization, potentially degrading image quality. Conversely, higher k preserves more visual detail but at the cost of larger file size.

5) Single-Threaded Parallel = Serial in Output

- Running the OpenMP implementation with 1 thread should the same output and behaviour as a fully serial version, confirming that parallelization is very much effective.

6) Optimal Balance Between Quality, Size, and Speed

- Values like k = 64 or 128 offer a practical sweet spot — offering good visual fidelity, reasonable compression (~47%–38%), and strong execution speedup with parallelism.

Challenges Faced:

1) Environment Setup and Compatibility Issues

Configuring OpenMP support in GCC under Windows required careful installation and path setup, especially with existing old compiler present which didn't support OpenMP and was already configured. Version mismatches and missing libraries (e.g., -lpthread, OpenMP flags) initially led to many difficulties.

2) Image Handling and Library Integration

Integrating stb_image and stb_image_write for loading and saving PNG images introduced challenges related to linking and unresolved references (stbi_load, stbi_write_png, etc.). Initially only ppm files were compatible, which is not user friendly.

3) Thread-Safe Parallelization

Ensuring that the K-Means clustering logic worked correctly in a parallel context required attention to data sharing, synchronization, and avoiding race conditions. Achieving both correctness and performance involved restructuring loops and using OpenMP pragmas effectively.

4) Performance Benchmarking and Automation

Writing a Bash script to automatically run the program across multiple thread counts and log the results involved careful control over environment variables and program arguments. Managing file naming, execution timing, and consistent logging was a tedious but necessary part of the workflow.

5) Data Visualization Difficulties

Generating clear, readable graphs (especially with overlapping labels) required learning plotting libraries and tweaking graph aesthetics. Switching between Python and R based on available packages and visualization quality added complexity to the analysis phase.

6) Debugging

Visual artifacts or incorrect cluster outputs sometimes appeared due to bugs in centroid updates or pixel assignment logic. Debugging in C without high-level tools made it difficult to trace logic errors or memory issues.

Conclusion:

This project successfully demonstrates the effectiveness of parallel computing in accelerating K-Means based image compression using OpenMP. By leveraging multithreading, significant reductions in execution time were achieved across all tested configurations, particularly at higher cluster counts (k). The performance analysis revealed that while speedup improves with more threads, the benefits diminish beyond a certain point due to overhead and limited parallel workload in lower k scenarios.

Furthermore, the trade-off between compression ratio and image quality was clearly evident. Lower k values resulted in higher compression rates with reduced file sizes, while higher k values preserved more visual detail at the cost of increased storage. Through this balance, cluster sizes like k = 64 and k = 128 emerged as optimal configurations, providing efficient compression with high fidelity and strong performance scaling.

Overall, the project highlights how parallelization not only accelerates computationally intensive tasks like K-Means clustering, but also enables practical real-time applications in image processing where performance and output quality must be carefully balanced.