PRO04: OpenMP k-means Clustering

Anthony Merlin

October 2024

1 Sequential k-Means

1.1 Problem Statement

Implement a C program that modify's the given source file kmeans_fun.c to approximate a solution to the k-means clustering problem using Lloyd's algorithm with initial cluster centers provided by the farthest first algorithm.

1.2 Implementation

1.2.1 Main Data Structures and Functions

The implementation consists of two primary functions that implement Lloyd's algorithm for k-means clustering:

find_clusters Function This function implements Step 2 of Lloyd's algorithm, assigning each point to its nearest cluster:

Listing 1: find_clusters implementation

```
void find_clusters(double* data, int len, int dim,
                   double* kmeans, int k, int* clusters) {
    for (int i = 0; i < len; i++) {
        double min_dist_sq = DBL_MAX;
        int closest_cluster = 0;

        for (int j = 0; j < k; j++) {
            double dist_sq = vec_dist_sq(data + i*dim,
                                         kmeans + j*dim, dim);
            if (dist_sq < min_dist_sq) {
                min_dist_sq = dist_sq;
                closest_cluster = j;
            }
        }
        clusters[i] = closest_cluster;
    }
}
```

The function works as follows:

- Iterates through each data point (outer loop with index i)
- For each point, calculates distance to all centroids (inner loop with index j)
- Uses squared Euclidean distance for efficiency (vec_dist_sq function)
- Tracks minimum distance and corresponding cluster index
- Assigns point to closest cluster by storing index in clusters array

1.3 Testing and Debugging find_clusters Function

- **Testing Strategy:**
  - Tested with small dataset (2-3 points) to verify correct cluster assignments
  - Verified handling of equidistant points to multiple centroids
  - Checked edge case of points exactly on centroids
- **Debugging Process:**
  - Used print statements to track distance calculations
  - Manually verified cluster assignments with small datasets
  - Checked correctness of pointer arithmetic in array access

calc_kmeans_next Function This function implements Step 3 of Lloyd's algorithm, computing the new centroid positions for each cluster:

Listing 2: calc_kmeans_next implementation

```
void calc_kmeans_next(double* data, int len, int dim,
                      double* kmeans_next, int k, int* clusters) {
    int *cluster_sizes = (int*) calloc(k, sizeof(int));
    if (cluster_sizes == NULL) {
        printf("Error: Memory allocation failed\n");
        exit(1);
    }

    // Initialize new centroids to zero
    for (int i = 0; i < k; i++) {
        vec_zero(kmeans_next + i*dim, dim);
    }

    // Accumulate points and count cluster sizes
    for (int i = 0; i < len; i++) {
```

```
        int cluster = clusters[i];
        cluster_sizes[cluster]++;
        vec_add(kmeans_next + cluster*dim,
                data + i*dim,
                kmeans_next + cluster*dim, dim);
    }

    // Calculate means and check for empty clusters
    for (int i = 0; i < k; i++) {
        if (cluster_sizes[i] == 0) {
            printf("Error: Empty cluster found\n");
            free(cluster_sizes);
            exit(1);
        }
        vec_scalar_mult(kmeans_next + i*dim,
                        1.0/cluster_sizes[i],
                        kmeans_next + i*dim, dim);
    }
    free(cluster_sizes);
}
```

The function implements centroid updates through three carefully designed phases:

1. Initialization Phase:

- Allocates dynamic memory for the cluster_sizes array using calloc
  - Uses calloc to ensure array is initialized to zeros
  - Includes error checking for failed memory allocation
- Initializes each dimension of every centroid to zero using vec_zero
  - Prepares kmeans_next array for accumulating point coordinates
  - Ensures clean start for centroid calculations

2. Accumulation Phase:

- Processes each data point exactly once
  - Retrieves the assigned cluster index
  - Increments the count for that cluster
  - Adds the point's coordinates to its cluster's sum
- Uses vec_add for efficient vector addition
  - Handles all dimensions of the point simultaneously
  - Accumulates sums for each dimension of the centroid

3. **Finalization Phase:**

- Performs crucial error checking
  - Detects empty clusters which would cause division by zero
  - Ensures proper cleanup on error conditions
- Computes final centroid positions
  - Divides accumulated sums by cluster sizes using vec_scalar_mult
  - Results in mean position for each cluster
- Manages memory cleanup
  - Properly frees the cluster_sizes array
  - Prevents memory leaks

1.4 Testing and Debugging calc_kmeans_next Function

- **Testing Strategy:**
  - Verified memory management using valgrind
  - Tested centroid calculations with simple, predictable datasets
  - Validated empty cluster detection and error handling
- **Debugging Process:**
  - Tracked cluster size counts and vector operations
  - Used GDB to step through memory allocation
  - Verified proper cleanup in error conditions

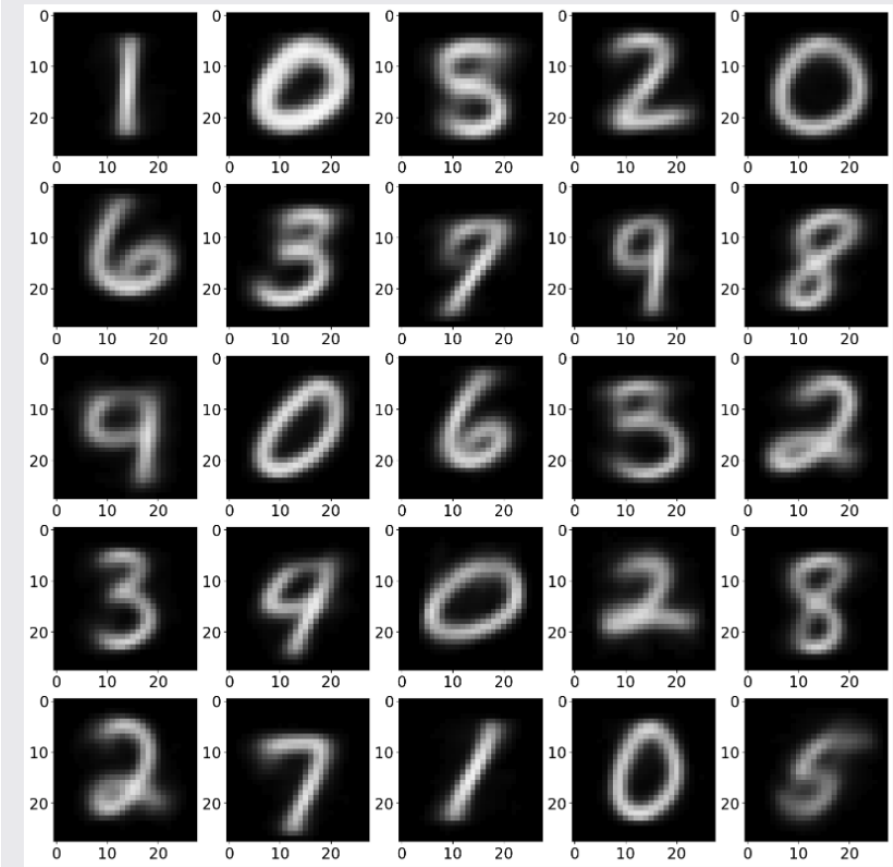
1.5 Demonstration Tasks

Here is the output of the program with k = 25 and num_iter = 55

```
(base) anthonym21@matrix:~/cmda3634/PR004$ gcc -o kmeans kmeans.c kmeans_fun.c vec.c
(base) anthonym21@matrix:~/cmda3634/PR004$ time cat mnist10k.txt | ./kmeans 25 55 > mnist10k_25_55.txt

real    0m38.357s
user    0m38.259s
sys     0m0.111s
(base) anthonym21@matrix:~/cmda3634/PR004$
```

and here is the screenshot of the .png image using Python showing graphically the results of applying kmeans with k = 25 and num_iter = 55.



2 OpenMP Farthest First

2.1 Problem Statement

The second part of the assignment required implementing a parallel version of the farthest first algorithm using OpenMP, specifically focusing on parallelizing the calc_arg_max function. The implementation needed to be both thread-safe and efficiently parallel.

2.2 Implementation

2.2.1 Parallel calc_arg_max Function

The parallel implementation of calc_arg_max uses OpenMP to distribute the workload across multiple threads:

Listing 3: Parallel calc_arg_max implementation

```
int calc_arg_max (double* data, int len, int dim, int* centers, int m) {
    int global_arg_max = 0;
    double global_cost_sq = 0.0;

    #pragma omp parallel
    {
        int local_arg_max = 0;
        double local_cost_sq = 0.0;

        #pragma omp for
        for (int i = 0; i < len; i++) {
            double min_dist_sq = DBL_MAX;

            for (int j = 0; j < m; j++) {
                double dist_sq = vec_dist_sq(data+i*dim,
                                             data+centers[j]*dim,dim);
                if (dist_sq < min_dist_sq) {
                    min_dist_sq = dist_sq;
                }
            }

            if (min_dist_sq > local_cost_sq) {
                local_cost_sq = min_dist_sq;
                local_arg_max = i;
            }
        }

        #pragma omp critical
        {
            if(local_cost_sq > global_cost_sq) {
                global_cost_sq = local_cost_sq;
                global_arg_max = local_arg_max;
            }
        }
    }
    return global_arg_max;
}
```

Key Implementation Features:

- **Parallel Strategy:**
  - Uses local variables for each thread to avoid race conditions
  - Parallelizes the outer loop for processing multiple points simultaneously

  - Employs critical section only for final global comparison

- **Thread Safety:**
  - Local variables (local_arg_max, local_cost_sq) prevent write conflicts
  - Critical section ensures thread-safe updates to global values
  - Read-only access to shared data (data, centers arrays)

2.3 Testing and Debugging

- **Correctness Testing:**
  - Compared results with sequential implementation
  - Verified consistency across different thread counts
  - Tested with the MNIST dataset (k=25, num_iter=0)

- **Performance Testing:**
  - Measured execution times with varying thread counts
  - Analyzed scaling efficiency using omp_kmeans_timing.sh
  - Verified performance improvement over sequential version

- **Debug Challenges:**
  - Initially faced race conditions in global value updates
  - Resolved by implementing local variables and critical section
  - Verified thread safety using multiple test runs

2.4 Conceptual Questions and Demonstration Tasks

1. Thread-Safe Implementation of Shared Variables: To ensure thread safety and prevent read-write race conditions in calc_arg_max, I implemented the following measures:

- **Local Variables:** Created thread-private variables for intermediate calculations:
  - local_arg_max: Stores the maximum argument for each thread
  - local_cost_sq: Stores the maximum cost for each thread

- **Shared Variables Protection:** Protected shared variables (global_arg_max and global_cost_sq) by:
  - Only updating them within a critical section
  - Ensuring read-only access to input arrays (data and centers)

2. Efficient Parallel Execution: To ensure efficient parallel execution with minimal critical section entry, I implemented:

- **Critical Section Usage:**
  - Each thread only enters the critical section once at the end of its parallel region
  - Critical section is placed outside the main computation loop
  - Only used for final global value updates
- **Work Distribution:**
  - Used #pragma omp for to distribute loop iterations across threads
  - Each thread processes its assigned points independently
  - Local computations are performed without synchronization

3. MNIST Clustering Results: Running the farthest first testing with k = 36 and num_iter = 0 produced the following visualization:

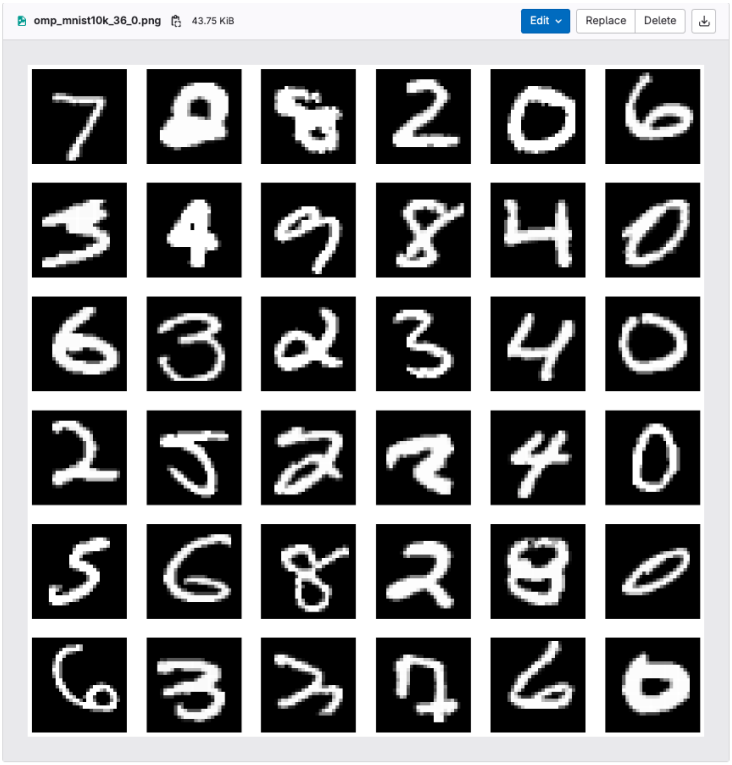


Figure 1: Result of k-means clustering on MNIST dataset (k = 36, num_iter = 0)

4. Performance Timing Results: The execution timing results for k = 36 and num_iter = 0 are shown below:

```
(base) anthonym21@matrix:~/cmda3634/PRO04$ bash omp_kmeans_timing.sh mnist10k.txt 36 0
(1,13.6329),(2,6.9014),(4,3.6313),(8,1.8827),(16,0.9763),(32,0.4836),
(base) anthonym21@matrix:~/cmda3634/PRO04$
```

Figure 2: Execution timing results with different thread counts

5. Strong Scaling Analysis: The strong scaling study results for k = 36 and num_iter = 0 are shown below:

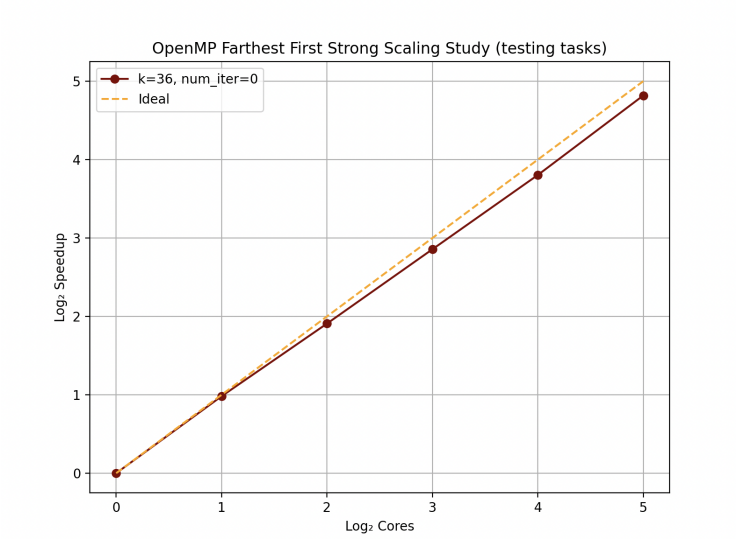


Figure 3: Strong scaling study for OpenMP Farthest First implementation (k = 36, num_iter = 0)

The plot shows our implementation achieves good scaling performance, with the actual speedup (maroon line) closely following the ideal speedup (orange dashed line).

3 OpenMP k-means

3.1 Problem Statement

Implement parallel versions of the k-means clustering functions using OpenMP, focusing on thread safety and efficient parallel execution. The implementation must avoid race conditions and minimize critical section usage while maintaining correctness.

3.2 Implementation

3.2.1 Main Data Structures and Functions

The parallel implementation consists of three primary functions:

calc_arg_max Function This function implements the farthest first algorithm in parallel as stated before.

find_clusters Function The parallel version of the cluster assignment function:

Listing 4: Parallel find_clusters implementation

```
void find_clusters (double* data, int len, int dim,
                    double* kmeans, int k, int* clusters) {
    #pragma omp parallel for
    for (int i = 0; i < len; i++) {
        double min_dist_sq = DBL_MAX;
        int closest_cluster = 0;

        for (int j = 0; j < k; j++) {
            double dist_sq = vec_dist_sq(data + i*dim,
                                         kmeans + j*dim, dim);
            if (dist_sq < min_dist_sq) {
                min_dist_sq = dist_sq;
                closest_cluster = j;
            }
        }
        clusters[i] = closest_cluster;
    }
}
```

Implementation features:

- Simple parallel for loop as iterations are independent
- Each thread processes its assigned points independently

3.3 Testing and Debugging

find_clusters Function:

- **Correctness Testing:**
  - Tested with small dataset (10 points, 2 dimensions) for manual verification
  - Compared outputs with sequential version to ensure identical results
  - Verified cluster assignments with different thread counts

- **Parallel Testing:**
  - Tested with varying number of OpenMP threads (1, 2, 4, 8, 16)
  - Verified thread independence using print statements
  - Confirmed no race conditions in cluster assignments
- **Debug Challenges:**
  - Verified correct point-to-cluster assignments across threads
  - Ensured proper array indexing with multidimensional data
  - Checked thread safety of vec_dist_sq function calls

calc_kmeans_next Function The parallel centroid update function:

Listing 5: Parallel calc_kmeans_next implementation

```
void calc_kmeans_next (double* data, int len, int dim,
                       double* kmeans_next, int k, int* clusters) {
    int *cluster_sizes = (int*) calloc (k, sizeof(int));
    double *local_sums = (double*) calloc (k * dim, sizeof(double));

    if (cluster_sizes == NULL || local_sums == NULL) {
        printf("Error: Memory allocation failed\n");
        exit(1);
    }

    #pragma omp parallel
    {
        int local_sizes[k];
        double local_means[k * dim];

        // Initialize local arrays
        for (int i = 0; i < k; i++) local_sizes[i] = 0;
        for (int i = 0; i < k * dim; i++) local_means[i] = 0.0;

        #pragma omp for
        for (int i = 0; i < len; i++) {
            int cluster = clusters[i];
            local_sizes[cluster]++;
            vec_add(local_means + cluster*dim, data + i*dim,
                    local_means + cluster*dim, dim);
        }

        #pragma omp critical
        {
            for (int i = 0; i < k; i++) {
```

```
                cluster_sizes[i] += local_sizes[i];
                vec_add(kmeans_next + i*dim, local_means + i*dim,
                        kmeans_next + i*dim, dim);
            }
        }
    }
}
```

Key parallel design features:

- Memory allocation outside parallel region
- Thread-local arrays for accumulation
- Single critical section per thread
- Efficient parallel accumulation of sums
- Thread-safe updates using local storage

3.4 Testing and Debugging

calc_kmeans_next Function:

- **Memory Management Testing:**
  - Used valgrind to check for memory leaks
  - Verified proper allocation/deallocation of cluster_sizes and local_sums
  - Tested error handling for allocation failures
- **Thread Safety Testing:**
  - Validated local array initialization in each thread
  - Confirmed proper accumulation in thread-local storage
  - Tested critical section updates with multiple threads
- **Debug Challenges and Solutions:**
  - Initially faced race conditions in cluster size updates
  - Resolved by implementing thread-local arrays
  - Added proper synchronization in critical section
  - Verified correct mean calculations across different thread counts

Integration Testing:

- Tested complete parallel pipeline:
  - Verified consistency with MNIST dataset (k=16, num_iter=40)
  - Compared results with sequential implementation
  - Validated final cluster centroids match expected patterns
- Performance Verification:
  - Measured speedup with different thread counts
  - Verified scaling efficiency
  - Profiled execution time of each component

3.5 Conceptual Questions and Demonstration Tasks

1. Thread-Safe Implementation for Shared Variables: To ensure thread safety and prevent read-write race conditions, I implemented the following measures:

- **calc_kmeans_next function:**
  - Used thread-local arrays (local_sizes and local_means) for intermediate calculations
  - Allocated shared memory (cluster_sizes and local_sums) outside parallel region
  - Protected shared data updates with a critical section
  - Each thread accumulates in its private arrays before combining results
- **find_clusters function:**
  - Each thread writes to independent locations in the clusters array
  - No shared variables are modified during computation
  - Input arrays (data and kmeans) are read-only

2. Efficient Parallel Implementation: To ensure efficient parallel execution with minimal critical section entry, I implemented:

- **Critical Section Usage:**
  - Each thread enters critical section only once after completing its computations
  - Critical section placed outside main computation loops
  - Updates to shared variables batched within single critical section

- **Work Distribution:**
  - Used #pragma omp parallel for to evenly distribute iterations
  - Maximized independent computation before synchronization
  - Local computations performed without locks or critical sections

3. MNIST Clustering Results: Running k-means with k = 25 and num_iter = 55 produced the following visualization:

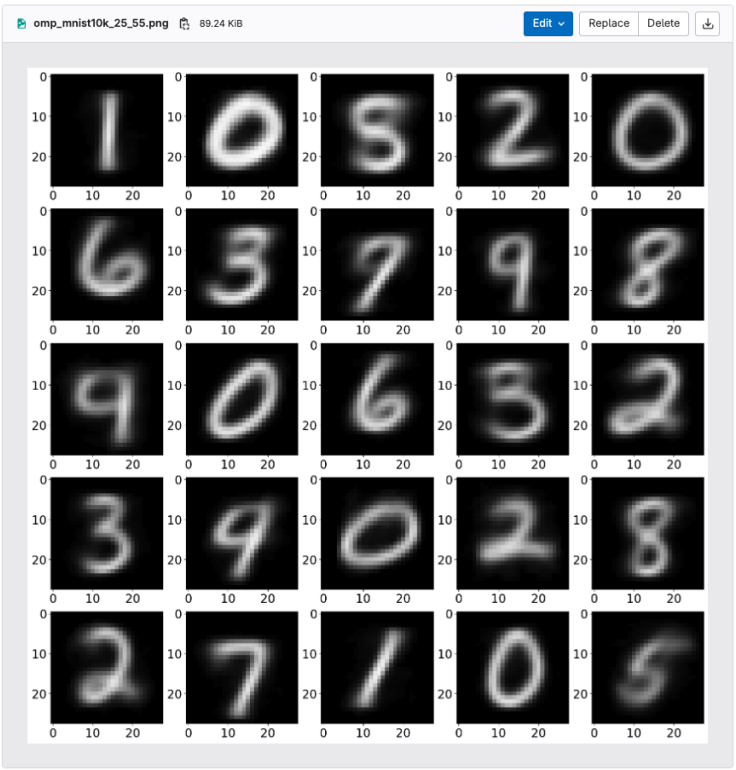


Figure 4: K-means clustering on MNIST dataset (k = 25, num_iter = 55)

4. Diff Command Output: Running diff command on the output images:

```
(base) anthonym21@matrix:~/cmda3634/PRO04$ diff omp_mnist10k_25_55.png mnist10k_25_55.png
(base) anthonym21@matrix:~/cmda3634/PRO04$
```

Figure 5: Output of diff command showing identical files

5. Timing Results: The execution timing results for k = 49 and num_iter = 5:

```
(base) anthonym21@matrix:~/cmda3634/PR004$ bash omp_kmeans_timing.sh mnist10k.txt 49 5
(1,30.6915),(2,15.4322),(4,8.1530),(8,4.2652),(16,2.1218),(32,1.0886),
(base) anthonym21@matrix:~/cmda3634/PR004$
```

Figure 6: Execution timing with k = 49 and num_iter = 5

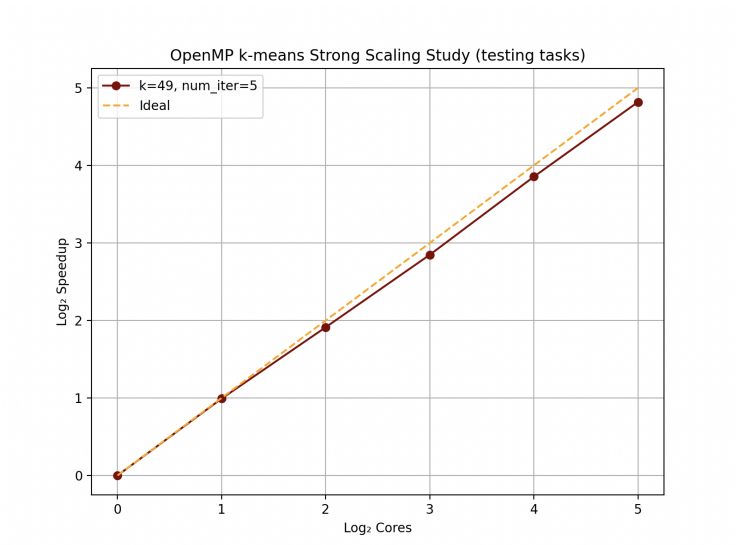


Figure 7: OpenMP k-means Strong Scaling Study (k = 49, num_iter = 5)

6. Strong Scaling Analysis: