Lab2: Parallelizing K-Means with OpenMP

Christos Karagiannis 3544 - Ioannis Sideris 3418 - Athanasios Gerampinis 3466
University of Thessaly – Department of Electrical and Computer Engineering

1 Introduction

This work studies the parallelization of the **K-Means** algorithm using the **OpenMP** framework in C. The goal was to accelerate clustering on large datasets while preserving correctness (identical centers and memberships compared to the sequential version).

The algorithm takes a set of N points with d dimensions and computes K clusters iteratively until convergence (maximum change below a given `threshold`).

2 Sequential Implementation

The sequential implementation lives in:

- `seq_kmeans.c` – main compute kernel (`seq_kmeans()`).
- `seq_main.c` – initialization, data I/O, and calling `seq_kmeans()`.

The main function, `seq_kmeans()`, repeats:

1. For each point, call `find_nearest_cluster(...)` (which performs the inner loop over all K centroids and computes distances).
2. Assign the point to the nearest centroid and update per-cluster sums and counts.
3. Recompute centroids as means of points in each cluster (repeat until the change ratio falls below `threshold` or a max-iteration cap).

These steps are compute-intensive, especially for large N, and are therefore prime targets for parallelization.

3 Parallelization Strategy

Parallelization was applied in the OpenMP version (`omp_kmeans.c`, `omp_main.c`) mainly at the following points:

3.1 Distance computation and cluster assignment

The inner double loop:

```
for (i = 0; i < numObjs; i++) {
    int index = find_nearest_cluster(numClusters, numCoords, objects[i], clusters);
    if (membership[i] != index) delta += 1.0;
    membership[i] = index;
    newClusterSize[index]++;
    for (j = 0; j < numCoords; j++)
        newClusters[index][j] += objects[i][j];
}
```

was parallelized with:

```
#pragma omp parallel for reduction(+:delta) schedule(static)
for (int i = 0; i < numObjs; i++) { ... }
```

so each thread processes a subset of the input points, accelerating the assignment stage.

3.2 Updating the new centroids

We handle centroid recomputation in a way that uses *per-thread privatization* with a parallel merge. We keep `schedule(static)` so iterations are split evenly once, with low overhead.

3.2.1 Privatization, merge and normalize

```
#pragma omp parallel shared(nthreads, partialClusterSize, partialClusters) \
                         reduction(+:delta)
{
  #pragma omp single { nthreads = omp_get_num_threads(); }
  #pragma omp for schedule(static)
  for (int i = 0; i < numObjs; i++) {
    int cid = find_nearest_cluster(...);
    localClusterSize[cid]++; /* + local sum to partialClusters */
    ...
}

#pragma omp parallel for schedule(static)         /* merge */
for (int c = 0; c < numClusters; c++) {
  int clusterCount = 0; /* + sum partials into newClusters[c][*] */
  ...
}

#pragma omp parallel for schedule(static)         /* normalize */
for (int c = 0; c < numClusters; c++) {
  if (newClusterSize[c] > 0) { float inv = 1.0f / newClusterSize[c]; ... }
  ...
}
```

3.3 Scheduling policy

Different `schedule` policies were tested:

- `static` (best performance for uniform load),
- `dynamic` (slightly higher overhead),
- `guided` (no notable improvement).

We finally used `schedule(static)`.

4 Optimizations Tried

We experimented with:

- **Parallel sections**: trying to parallelize assignment and recomputation as separate sections did not help, because the latter depends on the former.
- **collapse(2)**: applied to the distance loop, but brought no meaningful gains given the small dimensionality (`numCoords = 20`).

5 Compiler Flags

Compilation was driven by the `Makefile` with:

```
INCFLAGS    = -I.
OPTFLAGS    = -ffast-math -DNDEBUG
OMPFLAGS    = -qopenmp
```

Compiler: `Intel(R) oneAPI DPC++/C++ Compiler 2025.2.1 (2025.2.0.20250806)`.

6 Experimental Evaluation

Experiments were run on:

`Image_data/texture17695.bin`

using the following parameters:

- $N = 17695$ objects,
- $d = 20$ coordinates,
- $K = 2000$ clusters.

The sequential runtime was about **3.49 seconds**, while the OpenMP version with 56 threads reached **0.19 seconds**, i.e., **18.36**$\times$ speedup.

6.1 Results

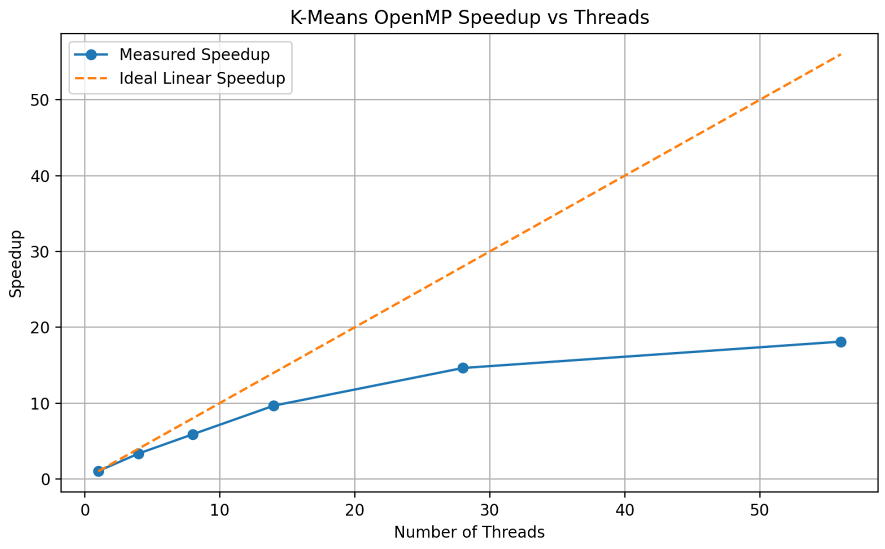

Figure 1: Speedup versus number of threads.

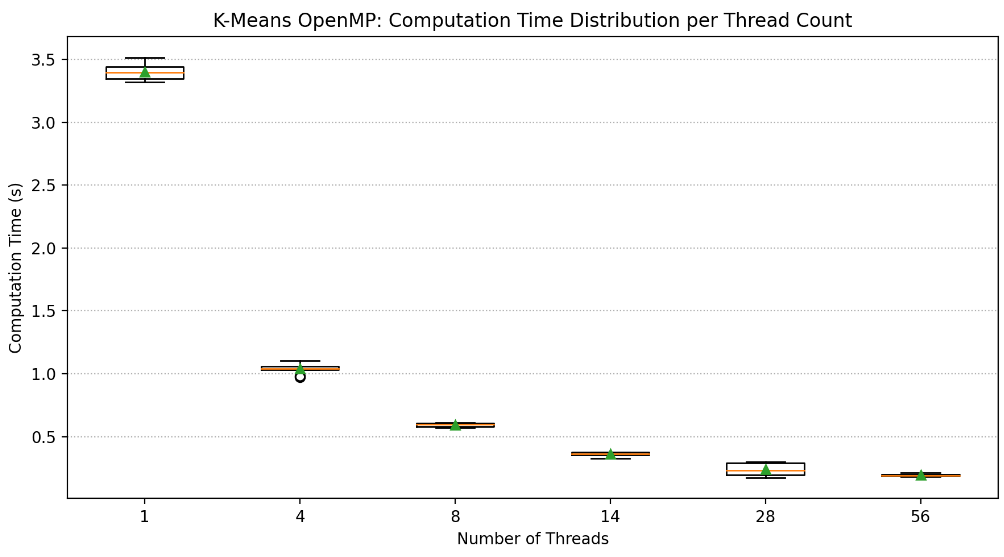

Figure 2: Computation time distribution per thread count.

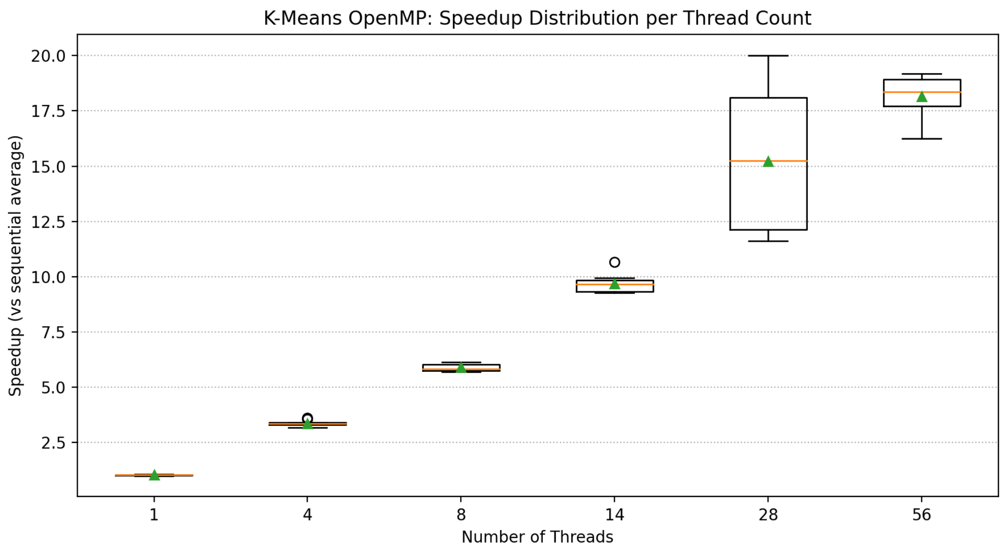

Figure 3: Speedup distribution per thread count.

6.2 Discussion

Scaling is good up to roughly 28 threads, close to linear. Beyond that, gains taper off due to:

- memory bandwidth saturation,
- reduced parallelism in the final iterations of K-Means.

The average speedup reaches 17.6× at 56 threads, with parallel efficiency around 31%.

7 Conclusions

OpenMP parallelization provided significant performance improvements on a multi-core system. Best performance appears when the number of threads matches the physical cores of the CPU.