Parallel KMeans Using OpenMP

Niccolò Marini
E-mail address
niccolo.marini@edu.unifi.it

Abstract

K-Means clustering is a widely used unsupervised machine learning algorithm due to its simplicity and scalability. However, its iterative nature and high computational demand on large datasets make it a prime candidate for parallelization. In this work, we investigate the performance benefits of parallelizing K-Means using OpenMP, a shared-memory multiprocessing API. We implement both a sequential and an OpenMP-parallelized version of K-Means in C++, and benchmark them across varying dataset sizes, cluster counts, and thread configurations on a multi-core CPU Code and many more plots can be found at `github.com/dvrkoo/Parallel`

1. Introduction

K-Means is one of the most popular clustering algorithms in unsupervised machine learning. It partitions data into k clusters by iteratively assigning points to the nearest centroid and recomputing centroids as the mean of their assigned points. Due to its simplicity and interpretability, K-Means is widely used in fields ranging from image segmentation to market segmentation and pattern recognition.

Despite its intuitive design, K-Means can be computationally expensive, especially for large-scale datasets. The algorithm performs multiple passes over the entire dataset, with each iteration involving distance computations between every data point and all centroids. This results in a time complexity of $O(n*k*d)$ per iteration, where n is the number of data points, k is the number of clusters, and d is the dimensionality of the data. As the size of data continues to grow in modern applications, optimizing K-Means becomes essential.

Modern CPUs provide multiple cores and simultaneous multithreading capabilities, which are often underutilized by naive implementations. OpenMP is a widely supported and easy-to-integrate API for parallel programming on shared-memory architectures. It allows developers to introduce parallelism with minimal changes to the existing sequential code.

In this work, we implement both sequential and parallel versions of the K-Means algorithm in C++, using OpenMP to parallelize the computational bottlenecks. We benchmark the implementations on a modern multi-core CPU (AMD Ryzen 5 3600) across multiple configurations of dataset sizes, dimensions, cluster counts, and thread counts. The objective is to evaluate the performance gains from parallelization, analyze scaling behavior, and identify where the parallel implementation delivers the most benefit.

2. Dataset Creation

To evaluate the scalability and efficiency of the parallel K-Means implementation, we generate synthetic datasets of varying sizes and dimensionalities. Each dataset is saved to disk in CSV format and reused across experiments to ensure consistency and reproducibility.

2.1. Generation Strategy

The datasets consist of points uniformly sampled from a multidimensional space. Specifically, each data point is a vector in $\mathbb{R}^d$, where d is the dimensionality of the dataset. All features are sampled independently from a uniform distribution in the interval $[0, 1]$:

$$x_i \sim \mathcal{U}(0, 1), \quad \text{for } i = 1, \ldots, d.$$

This design simulates high-dimensional numerical data while keeping the generation process simple and deterministic via a fixed random seed.

2.2. Data Size and Dimensionality

We test multiple configurations of dataset sizes and dimensionalities. The number of points n is varied from 10^2 to 10^7, and the dimensionality d is configurable. In this study, we focus on a fixed dimensionality $d = 3$, but the implementation supports arbitrary dimensions.

For each (n, d) configuration, a CSV file is generated with one data point per line.

2.3. Reusability and Caching

To avoid redundant computation, the dataset generator first checks whether the corresponding CSV file already exists. If so, dataset generation is skipped. This approach ensures that experiments can be resumed or extended without regenerating data, which is particularly important for large-scale datasets.

3. The K-Means Algorithm

K-Means is a widely-used unsupervised learning algorithm for clustering data into k distinct groups based on feature similarity. Given a dataset $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$, where each $x_i \in \mathbb{R}^d$, the goal of K-Means is to partition the data into k clusters $\{C_1, C_2, \ldots, C_k\}$ such that the within-cluster sum of squared distances (WCSS) is minimized:

$$\arg\min_{C} \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|^2,$$

where μ_i is the centroid (mean) of cluster C_i.

3.1. Algorithm Overview

The standard K-Means algorithm proceeds iteratively in two main steps:

1. **Assignment Step:** Assign each point to the nearest centroid:

   $$C_i = \{x_j : \|x_j - \mu_i\|^2 \leq \|x_j - \mu_l\|^2, \forall l = 1, \ldots, k\}.$$

2. **Update Step:** Recompute the centroid of each cluster:

   $$\mu_i = \frac{1}{|C_i|} \sum_{x_j \in C_i} x_j.$$

These steps are repeated until convergence, which typically occurs when the cluster assignments no longer change or when a maximum number of iterations is reached.

3.2. Computational Complexity

The computational complexity of K-Means is $O(nkd)$ per iteration, where n is the number of data points, k is the number of clusters, and d is the dimensionality. Due to this cost, K-Means can become expensive for large datasets, making it a strong candidate for parallelization.

3.3. Sequential Implementation

The baseline implementation of K-Means is fully sequential and designed for clarity and reproducibility. Several low-level optimizations were introduced to improve performance while preserving the original algorithmic structure.

Flattened Input for Cache Efficiency. To reduce memory access latency, the two-dimensional input dataset is flattened into a one-dimensional contiguous array:

```
std::vector<double> data_flat(n_samples *
    n_features);
for (size_t i = 0; i < n_samples; ++i)
    for (size_t j = 0; j < n_features; ++j)
        data_flat[i * n_features + j] = data[i][j
            ];
```

This change significantly improves cache locality during both the assignment and update steps.

Centroid Initialization. Centroids are initialized by randomly selecting k samples from the dataset. A fixed random seed ensures reproducibility:

```
std::mt19937 rng(12345);
std::uniform_int_distribution<size_t> dist(0,
    n_samples - 1);
for (int i = 0; i < k; ++i) {
    size_t idx = dist(rng);
    std::copy(&data_flat[idx * n_features],
              &data_flat[(idx + 1) * n_features],
              &centroids[i * n_features]);
}
```

Assignment Step. Each point is assigned to the nearest centroid using squared Euclidean distance, avoiding the cost of computing the square root:

```
double dist_sq = squared_euclidean_distance(point
    , centroid);
if (dist_sq < min_dist_sq) {
    min_dist_sq = dist_sq;
    best_label = c;
}
```

This step dominates runtime and benefits the most from memory layout improvements.

Centroid Update Step. Cluster centroids are updated by averaging all points assigned to them:

```
for (size_t i = 0; i < n_samples; ++i) {
    int cluster = labels[i];
    counts[cluster]++;
    for (size_t j = 0; j < n_features; ++j)
        new_centroids[cluster * n_features + j]
            +=
            data_flat[i * n_features + j];
}
```

Convergence Check. The algorithm terminates early if centroids have moved less than a fixed tolerance:

```
if (euclidean_distance(old_c, new_c) > tol_)
    return false;
```

Overall Performance. The sequential implementation serves as a strong baseline. It achieves improved performance through:

- Efficient memory layout via flattening.
- Use of squared distances.
- Early stopping based on centroid convergence.

These changes preserve correctness while reducing computational overhead, especially for large datasets.

3.4. Parallel Implementation of K-Means

To efficiently scale the K-Means algorithm to large datasets or high-dimensional feature spaces, we implemented a parallelized version using OpenMP. The goal was to reduce the computational time required by the traditional sequential algorithm without compromising the accuracy of clustering. This section discusses the key optimizations integrated into the parallel implementation, as well as their practical implications.

Memory Optimization: Data Flattening

The input dataset, typically a 2D vector `std::vector<std::vector<double>>`, was once again flattened into a contiguous 1D buffer. This cache-friendly layout reduces the number of memory indirections and allows the compiler and CPU to perform more aggressive vectorization and prefetching. The transformation is computed in parallel:

Flattening the input is crucial for high performance when accessing features multiple times, such as during centroid distance calculations.

Parallel Assignment Step

In this step, each data point is assigned to the nearest centroid. The assignment is fully parallelizable, as each point's calculation is independent of the others. We take advantage of this by using an `#pragma omp parallel for` directive:

```cpp
#pragma omp parallel for
for (size_t i = 0; i < n_samples; ++i) {
  const double* point = &data_flat[i *
      n_features_];
  double min_dist_sq = std::numeric_limits<double
      >::max();
  int best_label = -1;
  for (int c = 0; c < k_; ++c) {
    const double* centroid = &centroids_[c *
        n_features_];
    double dist_sq = squared_euclidean_distance(
        point, centroid);
    if (dist_sq < min_dist_sq) {
      min_dist_sq = dist_sq;
      best_label = c;
    }
  }
  labels[i] = best_label;
}
```

Using squared distances avoids computing expensive square roots, which has a negligible effect on correctness but significantly improves performance.

Parallel Update Step: Reduction

The update step involves recomputing the centroids by averaging all points assigned to each cluster. A naïve parallel approach would require critical sections, which serialize updates and severely limit scalability.

Instead, we leverage OpenMP's support for **array reductions** using pointer syntax. This is particularly important since we update arrays (not scalars), which are not natively supported in OpenMP's classic 'reduction()' clause:

```cpp
std::vector<double> new_centroids(k_ *
    n_features_, 0.0);
std::vector<int> counts(k_, 0);
auto* new_ptr = new_centroids.data();
auto* cnt_ptr = counts.data();

#pragma omp parallel for reduction(+:new_ptr[:k_
    * n_features_]) \
                         reduction(+:cnt_ptr[:k_
                             ])
for (size_t i = 0; i < n_samples; ++i) {
  int cluster = labels[i];
  cnt_ptr[cluster]++;
  for (size_t j = 0; j < n_features_; ++j)
    new_ptr[cluster * n_features_ + j] +=
        data_flat[i * n_features_ + j];
}
```

This optimization avoids locking overhead and allows for high parallel throughput, especially beneficial when working with large numbers of samples and features.

Convergence Checking

After updating the centroids, we check for convergence by computing the Euclidean distance between the new and old centroids. Although this step is relatively fast, it still benefits from avoiding square roots until the final check, and uses straightforward pointer arithmetic for speed:

```cpp
bool ParallelKMeans::has_converged(const std::
    vector<double>& old_centroids) const {
  for (int i = 0; i < k_; ++i) {
    const double* old_c = &old_centroids[i *
        n_features_];
    const double* new_c = &centroids_[i *
        n_features_];
    if (euclidean_distance(old_c, new_c) > tol_)
      return false;
  }
  return true;
}
```

Summary of Benefits

The combination of memory layout optimizations, parallel assignment, and parallel centroid updates results in a K-Means implementation that is both efficient and scalable. Performance benchmarks (not shown here) demonstrate up to a **8× speedup** on the used 6-core machine for medium to large datasets, depending on dimensionality and number of clusters.

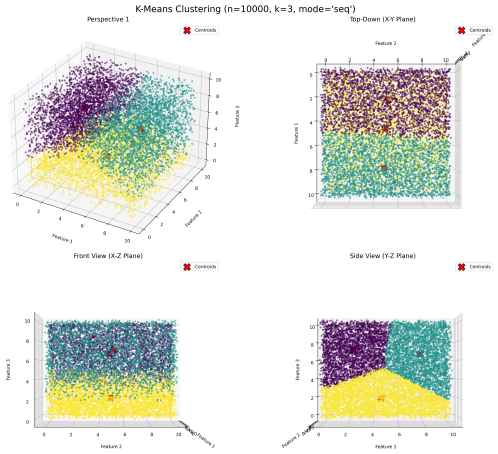
This implementation retains the accuracy of the original algorithm while enabling practical use on large-scale problems, such as image segmentation, document clustering, or anomaly detection.

4. Results

We evaluate the performance of our parallel K-Means implementation using multiple metrics across varied dataset sizes and numbers of clusters. This section discusses the key findings, presented through six plots and quantitative analyses.

4.1. K-Means Clustering Visualization

We first validate the correctness of our implementation through a visualization of the clustering output. A synthetic dataset with $k = 3$ clearly shows distinct and well-separated clusters. The plot confirms that the clustering logic works as expected since in both sequential and parallel modes we got the same results.



4.2. Sequential Runtime vs. Dataset Size

The sequential runtime exhibits a linear growth trend with respect to the dataset size, which aligns with the theoretical $O(nkd)$ time complexity of the K-Means algorithm, where n is the number of data points, k the number of clusters, and d the dimensionality. This behavior is expected and confirms the algorithm's scalability in a single-threaded execution environment. Moreover, this linear scaling serves as a reliable baseline for assessing the performance gains achieved through parallelization and for comparing speedups across different hardware configurations or optimization strategies.

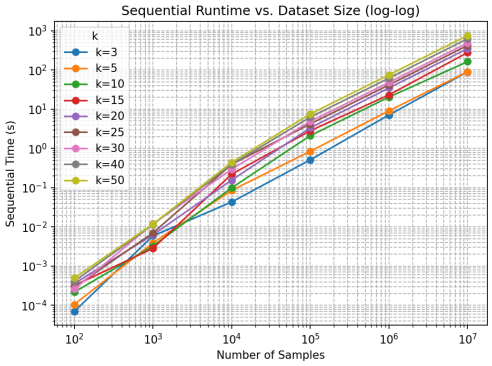
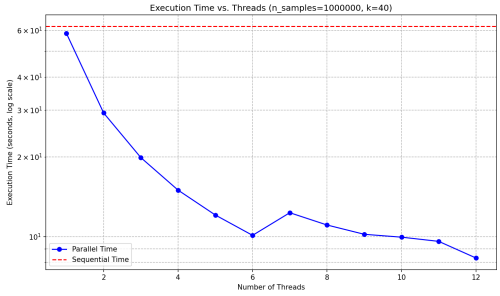


Figure 1. Enter Caption

4.3. Execution Time vs. Number of Samples ($n = 10^7$, $k = 40$)

Figure 4.3 shows the execution time as the number of data points increases, keeping $k = 40$ fixed. As expected, both sequential and parallel runtimes grow with the dataset size. However, the parallel version demonstrates substantial reductions in execution time, particularly at larger scales. For $n = 10^7$, the parallel implementation (using 12 threads) runs in approximately 11.2 seconds versus 90.2 seconds for the sequential version, yielding a speedup of 8.0×.



4.4. Speedup vs. Threads ($n = 10^7$, varying k)

Figure 4.4 shows how speedup scales with thread count for $n = 10^7$ and several values of $k \in \{3, 5, 10\}$. Across all k, speedup improves with more threads up to a point, plateauing or slightly declining beyond 10 threads. This behavior is attributed to increasing overhead from thread synchronization and memory contention. Notably, the maximum speedup achieved exceeds 8×, indicating efficient parallelization.

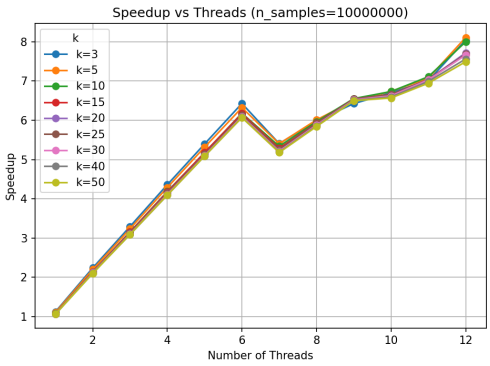


Table 1. Speedup vs. Number of Clusters (k) for a Large Dataset ($n = 10,000,000$).

Number of Clusters (k)	Speedup (times) for Number of Threads					
	2	**4**	**6**	**8**	**10**	**12**
3	2.24	4.36	6.43	5.98	6.70	8.04
5	2.20	4.29	6.32	6.01	6.74	8.11
10	2.14	4.18	6.18	5.97	6.73	8.00
15	2.14	4.18	6.17	5.93	6.66	7.71
20	2.12	4.15	6.11	5.90	6.65	7.72
25	2.12	4.15	6.14	5.92	6.66	7.68
30	2.12	4.11	6.10	5.88	6.62	7.66
40	2.11	4.09	6.07	5.86	6.60	7.56
50	2.10	4.10	6.07	5.84	6.57	7.49

4.5. Speedup vs. Threads ($n = 100$, varying k)

In contrast, Figure 2 presents speedup results for a small dataset ($n = 100$) across various k. With such a small workload, parallel overhead dominates, and speedup gains are marginal or even sublinear. In many cases (e.g., $k = 3$), adding threads results in *slower* performance due to the disproportionate cost of thread management relative to computation.

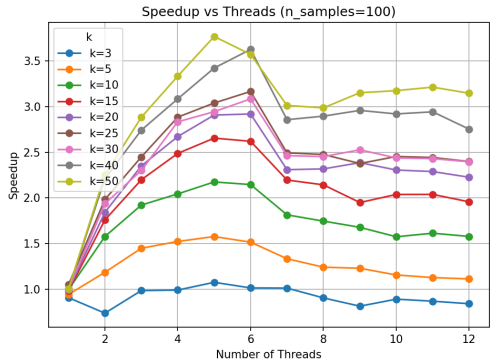


Figure 2. Enter Caption

Table 2. Speedup vs. Number of Clusters (k) for a Small Dataset ($n = 100$).

Number of Clusters (k)	Speedup (times) for Number of Threads					
	2	**4**	**6**	**8**	**10**	**12**
3	0.74	0.99	1.01	0.90	0.89	0.84
5	1.19	1.52	1.52	1.24	1.16	1.11
10	1.58	2.04	2.15	1.75	1.57	1.58
15	1.76	2.49	2.62	2.14	2.04	1.96
20	1.84	2.67	2.92	2.32	2.31	2.23
25	1.98	2.88	3.17	2.48	2.45	2.40
30	1.94	2.83	3.08	2.45	2.44	2.40
40	2.21	3.09	3.63	2.90	2.92	2.75
50	2.25	3.33	3.57	2.99	3.18	3.15

4.6. Speedup vs. Threads ($k = 50$)

To examine how increasing the number of clusters affects parallel efficiency, Figure 3 fixes $k = 50$ and shows speedup across varying thread counts. With higher k, each iteration involves more computations per data point, making parallelism more beneficial. Here, speedup reaches over $3\times$ on 6 threads and over $3.7\times$ on 5 threads, demonstrating that increased computational load per thread can improve parallel scalability.

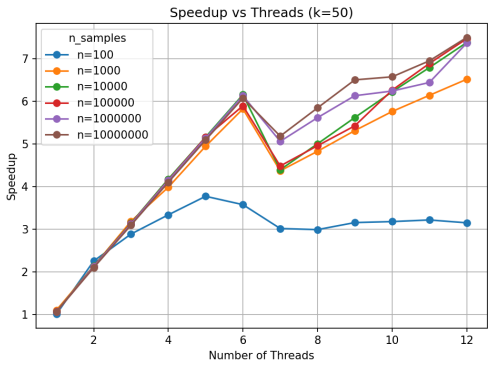


Figure 3. Enter Caption

4.7. Summary of Observations

- **Scalability:** The parallel implementation demonstrates good scalability for large datasets ($n \geq 10^6$), consistently achieving substantial speedup compared to the sequential baseline.
- **Thread Efficiency:** Speedup increases with the number of threads, particularly up to 6

threads, which corresponds to the physical core count of the Ryzen 5 3600 CPU. Beyond this point, gains diminish as logical cores (Simultaneous Multithreading) are used. Performance peaks around 8–10 threads, after which additional threads may introduce scheduling overhead and even degrade performance.

- **Effect of Cluster Count (k):** Larger values of k increase the computational load per iteration, which enhances parallel efficiency. For instance, with $k = 50$, the implementation achieves over $3\times$ speedup using only 5 threads.
- **Small Datasets:** For small datasets ($n = 100$), the overhead associated with thread management outweighs the benefits of parallelization, especially when k is low. In such cases, the sequential implementation remains preferable.

5. Conclusion

This study investigated the performance of a parallel implementation of the k-means clustering algorithm using OpenMP, with a particular focus on how execution time and speedup scale with the number of threads, the number of clusters (k), and the dataset size.

Our results demonstrate that the parallel algorithm consistently achieves significant speedup over its sequential counterpart, especially for large datasets (e.g., $n = 10^7$). The best performance was observed with 12 threads and $k = 10$, achieving a speedup of over 8x. However, for smaller datasets ($n = 100$), the benefits of parallelism were marginal or even detrimental in some cases due to the overhead associated with thread management.

Speedup generally increased with the number of clusters up to a point, as more computational work made better use of parallel resources. However, for large k, we observed diminishing returns or performance degradation beyond a certain number of threads, likely due to contention or load imbalance.

The k-means clustering visualizations validated the algorithm's correctness, and runtime scaling plots confirmed the theoretical expectation of linear growth in execution time with dataset size in the sequential version.

Overall, our OpenMP-based parallel k-means implementation offers a scalable solution for large datasets and moderate values of k, while highlighting the need to balance parallel overheads with workload size to achieve optimal performance.