A Comparative Analysis of AoS vs SoA in Parallel K-Means Implementation with OpenMP

Lorenzo Benedetti
lorenzo.benedetti5@edu.unifi.it

Abstract

This paper presents a detailed analysis of two different data layout strategies—Array of Structures (AoS) and Structure of Arrays (SoA)—applied to a parallel implementation of the K-means clustering algorithm. The study focuses on the performance implications of these memory layouts in both sequential and parallel contexts using OpenMP. We implemented and compared both approaches across different dataset dimensions, cluster sizes and number of features analyzing their impact on execution times, speedup, and overall computational efficiency. Our experimental results demonstrate the trade-offs between these approaches and provide insights into optimal data structure selection for parallel clustering implementations.

Future Distribution Permission

1. Introduction

In the era of big data and high-performance computing, efficient data clustering algorithms play a crucial role in data analysis and pattern recognition. K-means clustering, despite its simplicity, remains one of the most widely used algorithms in this domain. However, as dataset sizes continue to grow, the need for efficient implementations becomes increasingly important. This work explores the impact of data layout strategies on the performance of parallel K-means clustering implementations. We focus on two fundamental approaches to organizing data in memory: Array of Structures (AoS) and Structure of Arrays (SoA). While both approaches store the same information, their memory layout characteristics can significantly affect performance, particularly in parallel computing environments where memory access patterns and cache utilization become critical factors.

The primary contributions of this work include:

- A detailed comparison of AoS and SoA implementations for K-means clustering
- Analysis of parallel speedup and efficiency across different dataset dimensions
- Implementation considerations for OpenMP-based parallelization of clustering algorithms

2. K-Means

2.1. Description

K-means clustering is an unsupervised learning algorithm that aims to partition n observations into k clusters, where each observation belongs to the cluster with the nearest mean (centroid). The algorithm operates in multidimensional space, making it suitable for a wide range of applications including image processing, customer segmentation, and pattern recognition. The algorithm's objective is to minimize the within-cluster sum of squares (WCSS), also known as the inertia. Mathematically, given a set of observations $(x_1, x_2, ..., x_n)$, where each observation is a d-dimensional real vector, k-means clustering aims to partition the n observations into k sets $S = S_1, S_2, ..., S_k$ to minimize:

$$\sum_{i=1}^{k} \sum_{x \in S_i} |x - \mu_i|^2 \tag{1}$$

where μ_i is the mean of points in S_i.

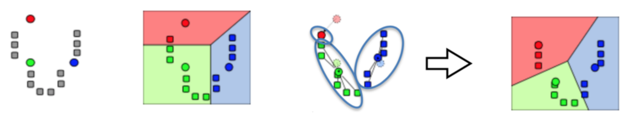

Figure 1. K-Means visualization

2.2. Algorithm

The K-means algorithm follows an iterative refinement approach:

Algorithm 1: K-Means Clustering

```
Input:
    • Dataset $X = \{x_0, \ldots, x_{\text{N\_SAMPLES-1}}\}$
    • N_SAMPLES
    • N_FEATURE_LIST list of dimensions
    • N_CLUSTERS
    • MAX_ITER number of epochs
    • NUM_THREADS list of threads
Output: Set of N_CLUSTERS
Assing each observation to the cluster with the
nearest centroid Calculate the Euclidean distance
repeat
    Calculate the new centroid of each cluster;
    Update the mean for each cluster;
until MAX_ITER is reached;
```

The algorithm is guaranteed to converge to a local optimum, though this may not be the global optimum. The quality of the final clustering depends significantly on the initial centroid selection, the number of clusters k and the number of epochs.

3. Implementation Details

In this section, we provide a detailed description of the implementation of our sequential and parallel k-means algorithms using two data layouts: Array of Structures (AoS) and Structure of Arrays (SoA). These layouts were chosen to evaluate their impact on performance, particularly with respect to memory access patterns and parallelization efficiency.

3.1. AoS

The Array of Structures (AoS) layout organizes data such that each point in the dataset is stored as a contiguous block in memory. Specifically, all attributes (dimensions) of a point are stored sequentially. This design benefits from improved locality of reference when accessing all dimensions of a single point but can lead to inefficient memory access during dimension-wise operations.

The `DatasetAoS` and `CentroidsAoS` structures encapsulate the AoS design. In the `DatasetAoS` implementation, each point is accessed using the `get_point` method, which retrieves a pointer to the memory block representing the point. Alternatively, the `at` method provides access to specific dimensions using the formula:

$$\text{Index} = \text{point\_index} \times \text{n\_dims} + \text{dim\_index}$$

Similarly, `CentroidsAoS` stores centroids as contiguous blocks and provides dimension-level access with the `at` and `get_centroid` methods. These structures were used to benchmark the AoS implementation in both sequential and parallel k-means.

3.2. SoA

In contrast, the Structure of Arrays (SoA) layout organizes data such that all values of a specific dimension are stored contiguously. This design improves memory access efficiency for dimension-wise operations but may result in scattered accesses when retrieving all attributes of a single point.

The `DatasetSoA` and `CentroidsSoA` structures represent the SoA design. Here, the `at` method provides access to a dimension of a specific point using the formula:

$$\text{Index} = \text{dim\_index} \times \text{n\_points} + \text{point\_index}$$

This approach minimizes cache misses during operations that process all points in a single dimension. The `CentroidsSoA` structure follows a similar pattern, ensuring efficient updates and retrievals during k-means iterations.

3.3. Compute Distance

The computation of distances between data points and centroids is fundamental to the k-means algorithm. Our implementation explores two memory layouts: Structure of Arrays (SoA) and Array of Structures (AoS), each with its own optimization strategies.

3.3.1 Memory Access Patterns

In the SoA layout, values for each dimension across all data points are stored contiguously:

```
const float* point_data = data.data.data();
const float* centroid_data = centroids.data.data
    ();

#pragma omp simd reduction(+:dist)
for (size_t d = 0; d < n_dims; ++d) {
    float diff = point_data[d * n_points +
    point_idx]
                  - centroid_data[d * k +
    centroid_idx];
    dist += diff * diff;
}
```

Listing 1. Compute Distance: SoA implementation

The AoS layout stores all dimensions of a single point together:

```
const float* point = data.get_point(point_idx);
const float* centroid = centroids.get_centroid(
    centroid_idx);

#pragma omp simd reduction(+:dist)
for (size_t d = 0; d < data.n_dims; ++d) {
    float diff = point[d] - centroid[d];
    dist += diff * diff;
}
```

Listing 2. Compute Distance: AoS implementation

SoA provides better cache locality for dimension-specific operations, while AoS optimizes access to complete points.

3.3.2 Euclidean Distance Computation

The implementation uses direct multiplication for squared differences instead of `std::pow` to reduce computational overhead:

```
float diff = point[d] - centroid[d];
dist += diff * diff;
```

Listing 3. Compute Distance: Euclidean Distance Computation

3.3.3 SIMD Parallelization

The implementation leverages Single Instruction, Multiple Data (SIMD) parallelism using OpenMP directives. The `pragma omp simd reduction(+:dist)` directive enables vectorized computation with these key aspects:

1. **Vectorization**: The compiler transforms loop operations into SIMD instructions, processing multiple data elements simultaneously.
2. **Reduction Process**:
   - Each thread maintains a private copy of the `dist` variable
   - Partial results are computed independently
   - Final results are combined using the specified reduction operation
3. **Thread Safety**: The reduction clause prevents race conditions by managing concurrent access to shared variables.

This approach significantly improves performance while maintaining computational accuracy, especially for high-dimensional datasets.

3.4. Sequential K-means

The sequential implementation of K-means follows the standard algorithm with two main phases: assignment and update. The implementation is templated to work with both Array of Structures (AoS) and Structure of Arrays (SoA) data layouts, adapting its behavior through compile-time specialization.

The algorithm iterates for a fixed number of iterations (`MAX_ITER`), and in each iteration:

1. For each data point, it computes distances to all centroids using the optimized `compute_distance` function, which employs SIMD instructions through OpenMP's `#pragma omp simd` directive.

2. During the update phase, it accumulates new centroid positions in a temporary buffer (`new_centroid_data`) while maintaining separate counters for each cluster.

3. Finally, it normalizes the centroids by dividing the accumulated values by their respective cluster counts.

The implementation leverages C++ templates to handle both data layouts efficiently:

- For SoA, data access patterns are optimized for vectorization, accessing contiguous memory locations for each dimension
- For AoS, the implementation uses direct pointer access through `get_point()` and `get_centroid()` methods to minimize indirection overhead

3.5. Parallel K-means

The parallel implementation extends the sequential version by introducing several OpenMP directives and optimizations to efficiently distribute the workload across multiple threads. The implementation includes careful consideration of data sharing and synchronization to maintain correctness while maximizing performance.

Key parallel implementation features include:

1. **Parallel Region Setup:**
   - Uses `#pragma omp parallel default(none)` to explicitly declare all variable sharing
   - Shares read-only data structures (`data`, `centroids`) and output vectors (`assignments`, `new_centroid_data`)
   - Makes `max_iter` firstprivate to ensure each thread has its own copy
2. **Local Accumulation Strategy:**
   - Each thread maintains private copies of centroid data and counts
   - Uses local buffers to avoid false sharing and reduce synchronization overhead
   - Merges local results only once per iteration using a critical section
3. **Work Distribution:**
   - Employs `#pragma omp for schedule(static)` for the main computation loop
   - Static scheduling chosen to maximize cache efficiency due to predictable access patterns
   - Combines assignment and accumulation phases to reduce memory traversals
4. **Data Layout Specific Optimizations:**
   - **SoA Layout:**
     - Direct access to underlying data arrays using `data.data()`
     - Enables better vectorization through contiguous memory access
     - Uses dimension-major ordering for improved cache locality
   - **AoS Layout:**
     - Leverages `get_point()` for efficient point access
     - Maintains point-major ordering for locality within points
     - Optimizes for cases where all dimensions are processed together
5. **SIMD Optimization:**
   - Uses `#pragma omp simd` for inner loops processing dimensions
   - Enables vectorization for distance computation and centroid updates
   - Combines with reduction for accurate accumulation of distances
6. **Synchronization Strategy:**
   - Uses `#pragma omp critical` for merging local results
   - Preferred over atomic operations due to better performance with multiple updates
   - Centralizes synchronization to minimize overhead

The final centroid normalization step is kept sequential as its computational cost is negligible compared to the main processing loop and parallelizing it would introduce unnecessary overhead.

3.6. Testing Infrastructure

To facilitate comprehensive testing and analysis of our K-means implementations, we developed a Python testing framework that automates dataset generation, experiment execution, and result visualization. The framework manages the execution of the C++ K-means implementations while providing tools for dataset creation and performance metric visualization.

3.6.1 Dataset Generation

The framework utilizes scikit-learn's `make_blobs` function to generate synthetic datasets with controlled characteristics. For our experiments, we generated datasets with the following parameters:

```
x, y = make_blobs(
    n_samples=1000000,    # 1M data points
    n_features=[2,3,4,8,16],   # Different
    dimensionalities
    centers=32,           # Number of clusters
    random_state=42       # For reproducibility
)
```

This approach allows us to test our implementations across different data dimensionalities while maintaining consistent cluster characteristics. The generated datasets are saved in CSV format for subsequent use by the C++ implementation.

3.6.2 Experiment Execution and Visualization

The Python framework automates the execution of experiments across different thread counts:

- 1, 2, and 4 threads on a M1 MacBook Air
- 1, 2, 4, 8, 16, 32, 48 threads on an AMD Ryzen Threadripper 2970wx

The experiments are also executed with both implementation variants (AoS and SoA). For each configuration, it:

1. Executes the C++ implementation using subprocess:

```
cmd = [
    "./cmake-build-release/kmeans",
    input_path,
    str(k),                 # k=32 clusters
    str(max_iter),          # 30 iterations
    str(threads),
    impl_flag              # 0 for AoS, 1 for
    SoA
]
```

2. Collects timing data (measured using `omp_get_wtime()`) and generates two types of visualizations using matplotlib:
   - Speedup comparison plots showing the relative performance improvement with increasing thread count
   - Execution time comparisons between sequential and parallel implementations for both AoS and SoA variants

   The framework also generates clustering visualizations for 2D and 3D datasets, enabling visual verification of the clustering quality.

All results and visualizations are automatically saved in a structured directory for further analysis.

4. Experimental Results

This study primarily focuses on speedup as the key performance metric while also considering raw execution times to provide a comprehensive performance analysis. The speedup is calculated as the ratio between sequential and parallel execution times for both Array of Structures (AoS) and Structure of Arrays (SoA) implementations.

4.1. Performance Metrics

Our experimental evaluation encompasses several dimensions of analysis:

- Implementation approach (AoS vs. SoA)
- Dataset dimensionality (varying from 2D to high-dimensional data)
- Number of data points (scaling from small to very large datasets)
- Number of clusters (varying k-means parameter)

- Thread count impact (scaling from single-thread to many-thread execution)

The experiments were conducted on two distinct hardware platforms to evaluate both consumer-grade and mid to high-performance scenarios:

- Apple M1 MacBook Air: Used for baseline performance testing, utilizing 4 performance cores. While the M1 chip features 8 cores in total (4 performance + 4 efficiency), we deliberately restricted our tests to the performance cores as the efficiency cores' significantly lower clock speeds would skew the parallel processing results.
- AMD Ryzen Threadripper 2970WX: Employed for scalability testing, featuring 24 cores and 48 threads, allowing us to analyze the implementation behavior under high thread counts. This platform enables investigation of potential scalability limitations and threading overhead effects.

For each configuration, we measure and compare:

- Sequential execution time (T_{seq})
- Parallel execution time (T_{par})
- Speedup ratio ($S = T_{seq}/T_{par}$)

All measurements are averaged over multiple runs (10) to ensure statistical significance, with both implementations tested under identical conditions to maintain fair comparison. The following sections present detailed analyses of these measurements, with particular emphasis on the scalability characteristics of both implementations across different problem dimensions and hardware configurations.

4.2. Speedup Analysis

This section presents a detailed analysis of the speedup achieved by both Array of Structures (AoS) and Structure of Arrays (SoA) implementations across various experimental dimensions. All experiments were conducted with a maximum of 30 iterations to ensure convergence. Unless otherwise specified, tests were performed using 100,000 data points, 32 clusters and 2 dimensions.

4.2.1 Impact of Data Points

Threads / Points	1	2	4
1'000	0.889	0.556	0.307
10'000	0.782	1.329	1.598
100'000	0.802	1.516	1.738
1'000'000	0.814	1.545	2.961
10'000'000	0.815	1.585	2.899
100'000'000	0.795	1.566	**2.966**

Table 1. Speedup for AoS implementation with varying number of points and threads.

The impact of dataset size on speedup was evaluated by varying the number of points from 1,000 to 100 million, using 16 clusters. Both implementations show similar scaling patterns, but with notable differences in their efficiency:

Threads / Points	1	2	4
1'000	0.831	0.862	0.526
10'000	0.805	1.276	1.626
100'000	0.843	1.607	1.803
1'000'000	0.854	1.654	3.070
10'000'000	0.820	1.636	3.163
100'000'000	0.849	1.674	**3.240**

Table 2. Speedup for SoA implementation with varying number of points and threads.

- For small datasets (1,000 points), both implementations exhibit poor speedup (< 1), indicating that parallelization overhead outweighs potential benefits.
- Performance improves significantly for datasets larger than 10,000 points, where both implementations achieve positive speedup.
- The SoA implementation consistently outperforms AoS, with a maximum speedup of 3.240x vs 2.966x with 4 threads for 100 million points.

- The performance gap between SoA and AoS widens as the dataset size increases, suggesting better cache utilization in the SoA implementation.

4.2.2 Impact of Dimensionality

Dataset dimensionality significantly influences the parallel performance of both implementations:

Threads Dim	1	2	4
2D	0.803	1.480	2.753
3D	0.937	1.791	3.298
4D	0.955	1.630	1.563
8D	0.934	1.791	2.047
16D	0.985	1.835	3.454
32D	0.977	1.908	3.497
64D	1.044	1.958	**3.732**
128D	0.959	1.859	3.323

Table 3. Speedup for AoS implementation with varying dimentionality.

Threads Dim	1	2	4
2D	0.854	1.599	2.927
3D	0.834	1.586	2.968
4D	0.970	1.632	1.904
8D	0.902	1.854	3.448
16D	0.988	1.937	**3.689**
32D	0.908	1.753	3.289
64D	0.951	1.693	3.588
128D	0.942	1.872	3.404

Table 4. Speedup for SoA implementation with varying dimentionality.

- Both implementations show generally improved speedup with higher dimensions, reaching peak performance at different dimensional points.

- The AoS implementation achieves its best speedup (3.732x) at 64 dimensions.

- The SoA implementation reaches its maximum speedup (3.689x) at 16 dimensions.

- A notable performance dip occurs at 4 dimensions for both implementations (1.563x for AoS, 1.904x for SoA), suggesting a potential cache-related bottleneck at this specific dimensionality.

- The SoA implementation maintains more consistent performance across different dimensions, indicating better memory access patterns.

4.2.3 Impact of Cluster Count

The number of clusters (k) affects the computational workload and, consequently, the parallel speedup:

- Both implementations achieve their best performance with a small number of clusters, with AoS peaking at 3.823x and SoA at 3.587x with k=2.

- Performance stabilizes for k8, with both implementations maintaining speedups between 2.7x and 3.6x.

- The SoA implementation shows better scalability with increasing cluster counts, achieving 3.599x speedup at k=128 compared to 3.367x for AoS.

- The overhead of managing cluster assignments and centroid updates becomes more significant as k increases, but SoA's memory layout appears to handle this overhead more efficiently.

Threads Clusters	1	2	4
2	1.430	2.349	**3.823**
4	1.011	1.771	2.757
8	0.859	1.549	2.762
16	0.814	1.498	2.754
32	0.904	1.704	3.215
64	0.829	1.591	2.971
128	0.922	1.772	3.367

Table 5. Speedup for AoS implementation with varying number of clusters.

Threads / Clusters	1	2	4
2	1.458	2.551	3.587
4	1.007	1.838	3.001
8	0.904	1.648	2.922
16	0.954	1.591	2.944
32	0.949	1.791	3.328
64	0.934	1.739	3.043
128	0.935	1.923	**3.599**

Table 6. Speedup for SoA implementation with varying number of clusters.

4.2.4 Thread Scaling Analysis

Testing on the AMD Threadripper 2970WX platform reveals significant differences in scaling behavior:

- Both implementations show near-linear speedup up to 4 threads, with efficiency above 90
- The SoA implementation demonstrates superior scaling, reaching a maximum speedup of 26.630x with 48 threads.
- The AoS implementation achieves a maximum speedup of 20.609x with 48 threads, approximately 22.6
- Performance scaling begins to flatten after 24 threads, suggesting diminishing returns from additional threads.
- The SoA implementation maintains better efficiency at higher thread counts, indicating more effective use of the memory hierarchy and reduced cache coherency overhead.

Threads / Impl	1	2	4	8	16	24	48
AoS	0.987	1.943	3.811	6.037	8.830	13.640	20.609
SoA	0.998	2.005	3.903	7.861	12.759	19.453	**26.630**

Table 7. Speedup results for AoS and SoA implementations across different thread counts.

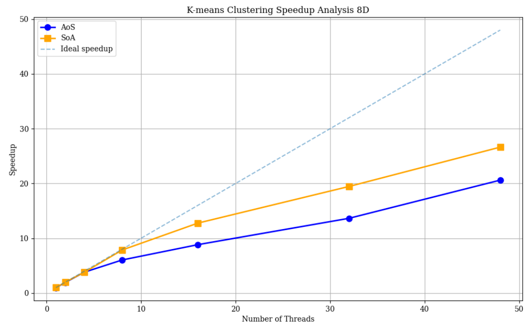

Figure 2. Thread scaling analysis

4.3. Execution Times

The execution time analysis for K-means implementations with 1M points, 32 clusters, and 2 dimensions reveals:

- For single-threaded execution, both implementations exhibit a slight overhead compared to their theoretical sequential performance, with execution times slightly higher than the sequential baseline.
- As the thread count increases, both implementations demonstrate significant performance improvements. The Structure of Arrays (SoA) implementation shows a more pronounced speedup, particularly with 4 threads, where it achieves a speedup of approximately 3.18x compared to the AoS implementation's 2.95x.

The most notable difference emerges in the parallel efficiency: the SoA implementation consistently demonstrates more efficient thread utilization, with lower parallel execution times and higher speedup ratios across different thread configurations. This performance advantage can be attributed to the SoA layout's superior memory access patterns and better cache locality, which become increasingly important as computational complexity and thread count increase.

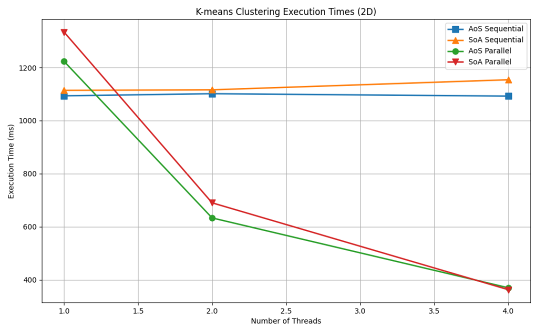

Figure 3. K-means execution times

5. Summary

This comprehensive analysis demonstrates that the SoA implementation generally provides better parallel scalability across all tested dimensions. The performance advantage becomes particularly pronounced with larger datasets and higher thread counts, suggesting that the SoA memory layout better suits modern CPU architectures' memory access patterns and cache utilization.

6. Appendix

6.1. Performance Analysis Challenges

Performance profiling presented unexpected challenges during the implementation. While Intel V-Tune would have been an ideal tool for identifying performance bottlenecks, particularly in investigating the performance variations between AoS and SoA implementations, several platform-specific constraints prevented its comprehensive deployment.

The primary obstacles emerged from hardware and software compatibility issues:

- Apple Silicon processors inherently restrict low-level performance analysis tools due to architectural and privacy considerations.

- On the EndeavourOS (Arch Linux) platform, V-Tune installation encountered significant package management conflicts. Specifically, conflicts arose between the standalone Intel V-Tune package, the Intel oneAPI Base Toolkit, and the system's `extra/intel-oneapi-basekit` package, rendering the standard installation procedures ineffective.

As for the software compatibility issues, it is important to note that Intel's documentation suggest using linux distributions that support the following package managers: APT, YUM, DNF, and Zypper.

As a pragmatic workaround, performance optimization was achieved through careful code refactoring, specifically by avoiding the use of the `at()` method to reduce overhead in the `compute_distance` function for the SoA implementation.

6.2. Source Code Availability

The complete source code for this K-means clustering implementation is publicly available and can be accessed through the GitHub repository at:
`github.com/lorenzo-27/kmeans`