Kmeans Clustering: Comparing Sequential and Parallel Implementations

Jay Senoner
E-mail address
jay.senoner@edu.unifi.it

Abstract

This report presents a comparative study between sequential and OpenMP-parallelized implementations of the k-means clustering algorithm. The aim is to assess the impact of parallelization on performance, particularly in terms of execution time and scalability, while preserving clustering accuracy. The parallel version employs OpenMP to accelerate the most computationally intensive phases of the algorithm: the assignment of data points to clusters and the update of cluster centroids. Results show that the OpenMP-parallel implementation achieves significant performance improvements over the sequential version on multi-core systems, especially for larger datasets, without compromising clustering quality. The findings underline the practical benefits and limitations of using shared-memory parallelism in iterative machine learning algorithms. Code is available at github.com/jaysenoner99/PP_1.

1. Introduction

In the field of data science and machine learning, the ability to find inherent groupings and structure within large datasets is a fundamental task. Clustering is a primary method of unsupervised learning used for this purpose, with applications ranging from customer segmentation in marketing and anomaly detection in cybersecurity to image compression and bioinformatics. One of the most widely known and frequently used clustering algorithms is KMeans, due to its conceptual simplicity, ease of implementation, and efficiency on a wide range of problems. The algorithm partitions a given dataset of n data points into k distinct, non-overlapping clusters by iteratively assigning each point to the cluster with the nearest mean, or "centroid", and then recalculating the centroid of each cluster based on its new members.

1.1. The Kmeans Algorithm

Formally, given a set of data points $(\mathbf{x}_1, \dots \mathbf{x}_n)$ where $\mathbf{x}_i \in \mathbb{R}^d \, \forall i \in [1, n]$, the k-means clustering algorithm aims to partition the data points into $k \leq n$ sets $\mathbf{S} = \{S_1, \dots, S_k\}$ in order to minimize the within-cluster sum of squares (i.e variance). The objective is to find:

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} ||\mathbf{x} - \boldsymbol{\mu}_i||^2$$

where $\boldsymbol{\mu}_i$ is the mean of points in S_i, often referred to as *centroid* of the cluster i:

$$\boldsymbol{\mu}_i = \frac{1}{|S_i|} \sum_{\mathbf{x} \in S_i} \mathbf{x}$$

Here $|S_i|$ denotes the cardinality of cluster i and $|| \cdot ||$ is the L2 norm. The algorithm operates iteratively: given a set of k initial centroids $\{m_1^{(1)}, ..., m_k^{(1)}\}$ the algorithm proceeds by alternating between two steps:

- **Assignment Step**: Assign each observation to the cluster with the nearest mean. Mathematically, this means partitioning the observations according to the Voronoi diagram generated by the means.

$$S_i^{(t)} = \left\{ x_p : ||x_p - m_i^{(t)}|| \leq ||x_p - m_j^{(t)}|| \;\; \forall j \in [1, k] \right\}$$

- **Update Step**: Recalculate centroids for observations assigned to each cluster.

$$m_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j$$

The time complexity of the kmeans algorithm is

$$\mathcal{O}(n \cdot k \cdot i \cdot d)$$

where n is the number of observations, k is the number of clusters, i is the number of iterations required for the algorithm to converge, and d is the dimensionality of the data points. It is important to note that since the within-cluster sum of squares decreases after each iteration, the sequence of objective function values is a non-negative monotonically decreasing sequence. This guarantees that the algorithm always converges, but not necessarily to the global optimum.

1.2. The case for Parallelization

The structure of the KMeans algorithm is inherently data-parallel. The most expensive step—calculating distances from every point to every centroid—can be performed independently for each data point. This makes the algorithm an ideal candidate for acceleration on modern multi-core processors using shared-memory parallel programming APIs like OpenMP. By distributing the workload across multiple CPU cores, it is possible to dramatically reduce the overall execution time, enabling the analysis of much larger datasets in a practical timeframe. Formally, given the time complexity $\mathcal{O}(n \cdot k \cdot i \cdot d)$ of the standard sequential implementation of the kmeans algorithm, we aim to achieve a time complexity of $\mathcal{O}(\frac{n}{P} \cdot k \cdot i \cdot d)$, where P is the available number of threads.

1.3. System Specification

All development, testing, and benchmarking were conducted on a desktop computer equipped with a 10th-generation Intel(R) Core(TM) i7-10700K CPU @ 3.80GHz. This processor features a monolithic architecture with 8 physical cores and supports Hyper-Threading, presenting a total of 16 logical cores (threads) to the operating system. All cores share a unified 16 MiB L3 cache. This specific hardware configuration is crucial to the performance analysis, as the distinction between physical and logical cores will be shown to have a direct and predictable impact on the scalability of the final optimized algorithm.

2. Implementation Details

This section details the implementations of both the sequential and parallel versions of the KMeans algorithm. A key focus is on establishing a fair, cache-aware sequential baseline to ensure that the measured speedup of the parallel version accurately reflects the gains from parallelization alone.

2.1. Data Structures and Dataset Generation

The *generateSyntheticDataset* method of the *Dataset* class creates a dataset of a specified size (`numExamples`) and dimensionality (`dimension`) with coordinate values uniformly distributed within a given range. `std::random_device` is used to seed the `std::mt19937` Mersenne Twister engine, a fast and robust pseudo-random number generator. `std::uniform_real_distribution` then maps the generator's output to the desired floating-point range, defined by the parameters [`minValue`,`maxValue`]. The generated points are stored in a `std::vector<Point>`, which serves as the primary data container for the *Dataset* class. To reduce the amount of time required to run each experiment, the dimension parameter of each observation in the dataset was set to 2. The design of the *Point* class is critical to the overall performance of the kmeans algorithm, as its instances are accessed millions of times within the core loops. For this project's focus on two-dimensional data, the point coordinates are stored in a `std::array<double, 2>` This choice offers several advantages: When a *Point* object is created, its coordinate data is allocated contiguously within the object's mem-

ory block. This improves data locality and cache performance, as fetching one coordinate from memory often brings the other into the same cache line. Moreover, unlike `std::vector`, `std::array` does not involve any dynamic heap allocation, eliminating pointer indirection and memory management overhead.

Performance-critical operations that are called frequently within loops are implemented as `inline` functions. This includes the distance calculation, getters/setters, and overloaded operators (+=, /). The overloaded operators provide an intuitive and efficient syntax for the vector arithmetic required in the centroid update step. The *reset()* method, which uses the highly efficient `std::array::fill`, is crucial for the zeroing-out of temporary storage.

2.2. The sequential baseline

The standard, sequential kmeans clustering algorithm was implemented within a custom *Kmeans* class. The clustering procedure is encapsulated in the *kMeansClustering* method, which iteratively updates cluster assignments and centroids for a fixed number of epochs or until convergence.

Centroids are initialized using the *random_choice* method, which selects k unique points uniformly at random from the dataset.

In each iteration, data points are assigned to the nearest centroid using the *min_distance_cluster* function. To enhance performance, a local cache-friendly copy of centroids is used.

```
int Kmeans::min_distance_cluster(
    const Point &p, const std::vector<Point
        > &current_centroids) const {
  double min_dist = std::numeric_limits<
      double>::max();
  int best_cluster = -1;

  for (size_t i = 0; i < current_centroids.
      size(); ++i) {
    double dist = current_centroids[i].
        distance(p);
    if (dist < min_dist) {
      min_dist = dist;
      best_cluster = static_cast<int>(i);
    }
  }
  return best_cluster;
}
```

By passing this local copy to *min_distance_cluster*, we ensure that all centroid data is accessed with minimal latency, preventing cache misses from becoming a performance bottleneck. The *min_distance_cluster* function itself is also optimized to avoid unnecessary memory allocations by directly iterating and comparing distances, rather than storing them in a temporary vector.

The update step calculates the new position for each centroid. This is implemented as a two-stage process:

- **Accumulation**: Two temporary vectors, `new_centroids` (of type *Point*) and `counters` (of type *int*), are created and initialized to zero. The code then iterates through all data points, summing their coordinates into the appropriate `new_centroids` entry and incrementing the corresponding counter.
- **Recalculation**: A final loop divides each summed coordinate vector in `new_centroids` by its corresponding count in `counters`, yielding the new mean position. This division is handled by the overloaded `operator/` in the Point class.

Convergence is checked after every epoch by calling the *not_changed* method, which checks whether the L2 norm difference between old and new centroids falls below a specified tolerance.

```
bool Kmeans::not_changed(const std::vector<
    Point> &new_centroids, double tol) {
  if (centroids.size() != new_centroids.
      size())
    return false; // Safety check
  for (size_t i = 0; i < centroids.size();
      ++i) {
    if (centroids[i].l2normdiff(
        new_centroids[i]) >= tol)
      return false;
  }
  return true;
}
```

At the end of each iteration, the newly computed centroids replace the previous ones. If convergence criteria are met or the maximum number of iterations is reached, these updated centroids are returned.

```
void Kmeans::kMeansClustering(int
    max_epochs, int k) {
  if (data.empty())
    return;

  centroids = random_choice(k);

  for (int epoch = 0; epoch < max_epochs;
      ++epoch) {
    const std::vector<Point>
        local_centroids = centroids;

    for (auto &data_point : data) {
      int new_cluster =
          min_distance_cluster(data_point,
          local_centroids);
      data_point.setCluster(new_cluster);
    }

    std::vector<Point> new_centroids(k);
    std::vector<int> counters(k, 0);

    for (const auto &data_point : data) {
      int cluster_id = data_point.
          getCluster();
      if (cluster_id != -1) {
        new_centroids[cluster_id] +=
            data_point;
        counters[cluster_id]++;
      }
    }

    for (int i = 0; i < k; ++i) {
      if (counters[i] > 0) {
        new_centroids[i] = new_centroids[i]
            / counters[i];
      }
    }

    if (not_changed(new_centroids)) {
      centroids = new_centroids;
      return;
    }

    centroids = new_centroids;
  }
}
```

3. Parallel Implementation

To improve the scalability and performance of the clustering algorithm on large datasets, we implemented a parallelized version of kmeans in the *ParallelKmeans* class using OpenMP for shared-memory parallelism. The core clustering logic is encapsulated in the *parallelKmeansClustering* method, which mirrors the sequential structure but introduces multithreading to parallelize the assignment and update phases.

The initial centroids are selected using the same *random_choice* function as in the sequential version. This ensures a uniform and reproducible initialization across both implementations.

Auxiliary data structures, such as `new_centroids` (to accumulate cluster means) and `cluster_counters` (to track the number of points per cluster), are initialized before entering the main loop. To prevent repeated memory allocation within the main loop, these vectors are allocated once and then reset to zero using the `Point::reset()` method and `std::fill()`.

The most computationally expensive part of KMeans is the **assignment step**. In a parallel setting, having multiple threads read from a single, shared `centroids` vector can lead to cache coherence traffic and reduced performance. To avoid this problem, a thread-local copy of the current centroids (`local_centroids`) is created and marked as `const`. This significantly improves cache locality and reduces memory contention, as threads avoid competing for reads on the shared centroids vector. When *min_distance_cluster* is called, it accesses this local copy, eliminating high-latency main memory traffic and ensuring the assignment step operates at maximum speed.

The **update step** presents a classic data race condition: multiple threads cannot safely update the same global `new_centroids` vector simultaneously. A naive solution using a single lock around each point's update would create a severe sequential bottleneck. Our implementation avoids this by using a manual reduction pattern.

Within the parallel region, each thread accumulates its results into its own private vectors: `private_centroids_sum` and `private_counts`. Since each thread writes only to its own memory, the main assignment and accumulation loop can proceed without any locks or synchronization, achieving maximum parallelism. The `nowait` clause on the `#pragma omp for` directive is a further optimization, allowing threads to proceed immediately without waiting at an implicit barrier. Since each thread operates on its own private data in this loop, there are no data dependencies or shared-memory conflicts that require synchronization at the loop boundary. Eliminating unnecessary synchronization reduces the overhead of context switching and waiting, especially in cases where workload distribution is even (as enforced here by `schedule(static)`). Only after all threads have completed this work are the private results combined into the global `new_centroids` and `cluster_counters` vectors. This final reduction step is protected by a single `#pragma omp critical section`. As this section is entered only once per thread per epoch, its overhead is negligible compared to the massive parallel workload of the main loop.

Finally, the **recalculation step**—dividing each summed centroid by its count—is also parallelized using a simple `#pragma omp parallel for`, as the update for each centroid is an independent operation.

Finally, the algorithm checks whether the centroids have stabilized using the *not_changed* function as in the sequential implementation. If convergence is detected (i.e. centroids have not changed significantly) or the maximum number of iterations is reached, the algorithm exits early. Otherwise, the updated centroids are used in the next iteration.

The number of threads can be configured via the `threads` argument, enabling flexible performance evaluation across different number of threads.

All other aspects of the algorithm (initialization, stopping criteria, etc.) remain consistent with the sequential version to ensure fair comparisons.

```
void ParallelKmeans::
    parallelkMeansClustering(int max_epochs,
     int k,int threads) {
  if (data.empty())
    return;

  centroids = random_choice(k);

  const int n = data.size();
  std::vector<Point> new_centroids(k);
  std::vector<int> cluster_counters(k, 0);

  for (int epoch = 0; epoch < max_epochs;
      ++epoch) {
    for (auto &p : new_centroids)
      p.reset();
    std::fill(cluster_counters.begin(),
        cluster_counters.end(), 0);

#pragma omp parallel num_threads(threads)
    {
      const std::vector<Point>
          local_centroids = centroids;

      std::vector<Point>
          private_centroids_sum(k);
      std::vector<int> private_counts(k, 0)
          ;

#pragma omp for schedule(static) nowait
      for (size_t j = 0; j < n; ++j) {
        // Pass the thread's FAST local
            copy to min_distance_cluster
        int cluster_id =
            min_distance_cluster(data[j],
            local_centroids);
        data[j].setCluster(cluster_id);
        private_centroids_sum[cluster_id]
            += data[j];
        private_counts[cluster_id]++;
      }

#pragma omp critical
      {
        for (int c = 0; c < k; ++c) {
          new_centroids[c] +=
              private_centroids_sum[c];
          cluster_counters[c] +=
              private_counts[c];
        }
      }
    } // --- End of parallel region ---

#pragma omp parallel for num_threads(
    threads)
```

```
    for (int j = 0; j < k; ++j) {
      if (cluster_counters[j] > 0) {
        new_centroids[j] = new_centroids[j]
            / cluster_counters[j];
      }
    }

    if (not_changed(new_centroids)) {
      centroids = new_centroids;
      return;
    }
    centroids = new_centroids;
  }
}
```

4. Performance Evaluation and Analysis

To rigorously evaluate the performance of the optimized parallel kmeans algorithm, a comprehensive testing framework was developed. This framework was designed to ensure the reproducibility of results and to facilitate a fair comparison between the sequential and parallel implementations across a range of problem sizes (defined in terms of values of n and k) and thread counts.

4.1. Experimental Setup

To quantify the performance gain obtained through parallelization, we specifically measured the **speedup** obtained through parallelization, defined as:

$$Speedup(P) = \frac{t_s}{t_P}$$

where t_s is the average execution time of the sequential algorithm and t_P is the average execution time of the parallel algorithm using P threads.

All timing measurements were performed using the high-resolution *omp_get_wtime()* function from OpenMP. To mitigate the effects of system jitter and ensure stable results, each test configuration was executed 50 times, and the average execution time was recorded.

The experimental procedure for a single problem size n and cluster count k is as follows:

- **Dataset Generation**: A synthetic 2D dataset of n points is generated via the *generateSyntheticDataset* function, with coordinates uniformly distributed between 0 (`minValue`) and 1000 (`maxValue`). This single dataset instance is used for both the sequential and subsequent parallel runs to ensure a direct comparison.
- **Sequential Baseline Measurement**: The sequential *kMeansClustering* method is executed 50 (`repetitions`) times. The total execution time is accumulated and then averaged to produce a stable sequential time, t_s
- **Parallel Performance Measurement**: A loop iterates through thread counts P from 2 up to the maximum available on the system (16). For each P, the *parallelkMeansClustering* method is executed 50 times, and the average parallel time, t_P, is calculated.
- **Speedup Calculation**: The speedup for each thread count P is calculated using the standard formula: $Speedup(P) = t_s/t_P$.

This process was automated to run across several problem sizes, specifically for values of n varied across several orders of magnitude, ranging from $n = 10^3$ to $n = 10^6$, increasing geometrically by a factor of 10 for each test, while k is being fixed. To analyze the impact of increasing the number of clusters k on performance gain, a range of progressively larger k values was tested: $k = 10, 20, 30, 40, 50, 100$

4.2. Benchmarking Code Overview

The benchmarking code is structured around two main routines:

- `run_test(n, k, max_epochs, repetitions)`: Executes one full test for a given dataset size and cluster count, computing the average runtime for both sequential and parallel versions, and returns the speedup results.
- `run_multiple_tests(max_n, k, max_epochs, repetitions, filename)`: Iteratively runs tests for increasing dataset sizes (up to `max_n`) and

stores the speedups in a CSV file in order to prepare them for visualization.

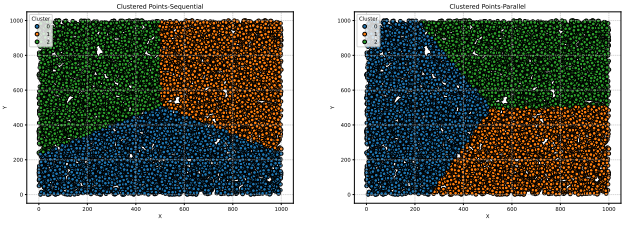
To visualize progress during execution, a helper function *print_progress* displays a real-time progress bar in the terminal.

5. Results

This section presents the results of comparing the optimized parallel implementation of the k-means algorithm with its sequential counterpart, using the experimental setup described in Section 4.1. These speedup measurements consistently highlight the effectiveness of the parallel approach and provide valuable insights into the interaction between algorithmic behavior, problem size, and the underlying hardware architecture.

5.1. Clustering Visualization

To qualitatively validate that the parallel implementation produces correct and comparable results to the sequential baseline, a visualization test was conducted. Both the sequential and the parallel algorithms (using 8 threads) were run on an identical synthetic dataset with $n = 10,000$ points and a target of $k = 3$ clusters. The final cluster assignments for each point were saved as a `.csv` file and plotted using a python script (named `plot_clusters.py`). The results of clustering visualization are shown in figure 1.



(a) Sequential clustering (b) Parallel clustering

Figure 1: Comparison of clustering results using sequential and parallel k-means.

The visualization confirms that the resulting clusters form valid Voronoi partitions over the square region of $\mathbb{R}^2$ spanning from 0 to 1000 along both the x and y axes. Variations in the clustering results are likely attributable to the random initialization of the cluster centroids, to which the kmeans algorithm is known to be sensitive.

5.2. Speedup analysis

Speedup measurements are shown in Figures 2 through 7. Across all tested values of k, several consistent trends emerge from the speedup measurements.

Strong Scalability on Physical Cores: For all problem sizes, the algorithm exhibits strong, near-linear speedup as the thread count increases from 2 to 8. This corresponds directly to the 8 physical cores of the Intel i7-10700K CPU used for testing. As each new thread is assigned to an idle physical core, it contributes its full computational power, leading to significant performance gains.

The Hyper-Threading Boundary at 8 Threads: A distinct and sharp performance inflection point occurs at $P = 8$ and $P = 9$ across all experiments. The speedup consistently peaks at 8 threads, often followed by a noticeable dip at 9 threads. This behavior marks the transition from utilizing physical cores to utilizing logical cores via Hyper-Threading. The 9th thread must share the resources of a physical core that is already fully engaged, leading to increased contention and overhead that temporarily negates the benefit of the additional thread.

Marginal Gains from Hyper-Threading: As the thread count increases from 9 to 16, performance generally recovers and continues to improve, but at a much shallower slope than in the 1-8 thread region. This demonstrates the modest benefit of Hyper-Threading for this compute-bound workload, providing a typical 15-25% additional performance by scheduling work during memory stalls, but falling far short of the gains provided by true physical cores. The maximum speedup achieved across all tests approaches approximately 8.5x, aligning with expectations for an 8-core processor with hyperthreading.

5.3. The impact of problem size n

The cardinality n of the generated dataset has an huge impact on performance gains, since the fraction of points managed by each thread increases as n increases. The speedup measurements on the conducted tests confirm that:

- **Higher n yields better speedups**: For any given thread count P, the speedup achieved is consistently higher for larger values of n. For example, in the test with $k = 50$ (Figure 6), the speedup at $P = 16$ is approximately 5.1x for $n = 10^3$, but reaches 8.4x for $n = 10^6$. This is because for larger datasets, the computational work of the parallel loops dominates the total execution time, making the fixed overhead of thread creation and synchronization comparatively insignificant.
- **Performance for Small n:** For the smallest dataset ($n = 1000$), the speedup is the most modest and erratic. This is expected, as the overhead of managing the parallel execution can be comparable to the actual computation time, limiting the overall efficiency.

5.4. The impact of cluster count k

The number of clusters k directly affects the computational workload per iteration and hence the efficiency of parallel execution. This is because increasing k increases the amount of work done per data point in the assignment step. While the overall shape of the scalability curve remains consistent, a detailed comparison of the plots reveals that increasing k leads to higher and more stable parallel efficiency. The stability is evident when comparing the relatively noisy behavior in the $k = 10$ plot (Figure 2) with the smooth, well-defined scaling curves in the $k = 100$ plot (Figure 7).

Moreover, runs with a lower value of n manage to get higher speedups when k is raised. For example, when $k = 10$ (Figure 2) and $n = 1000$, the speedup is $\approx$ 4x for $P = 16$, while in the $k = 100$ setting (Figure 7), with the same values of n and P, the obtained speedup is $\approx$ 5.3x. Interestingly, in the $n = 10^4$ setting, the speedup for $P = 16$ raises from $\approx$ 7x to $\approx$ 8.5x, surpassing the speedup values obtained for $n = 10^5$ and $n = 10^6$ with the same number of threads and cluster count.

Moreover, runs with a lower n value tend to achieve higher speedups as k increases. For instance, with $n = 1000$ and $P = 16$, the speedup improves from approximately $4\times$ when $k = 10$ to around $5.3\times$ when $k = 100$. Interestingly, in the $n = 10^4$ setting, the speedup for $P = 16$ increases from approximately $7\times$ to $8.5\times$ as k goes from 10 to 100, surpassing the speedup values observed for $n = 10^5$ and $n = 10^6$ under the same thread and cluster count.

5.5. Speedup Plots

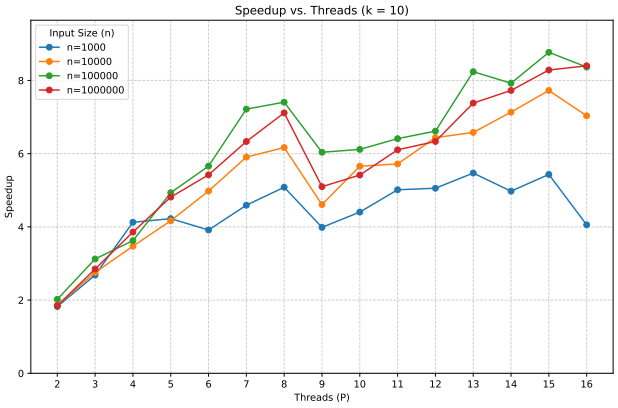


Figure 2: Speedup vs. Threads, k=10

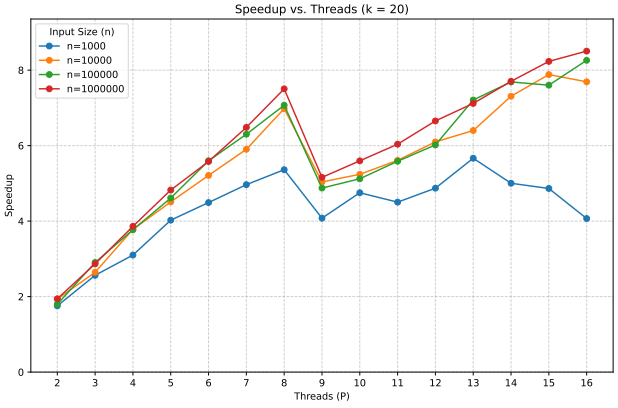


Figure 3: Speedup vs. Threads, k=20

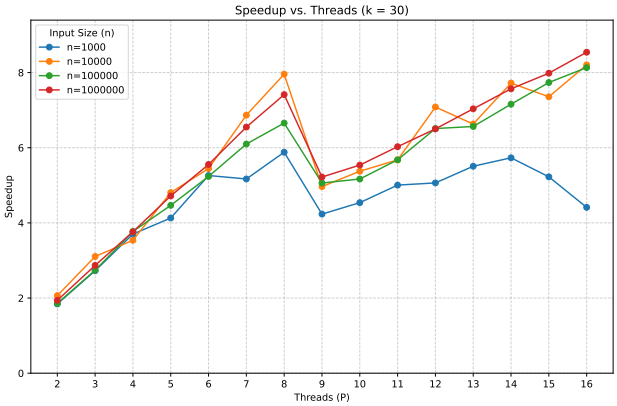


Figure 4: Speedup vs. Threads, k=30

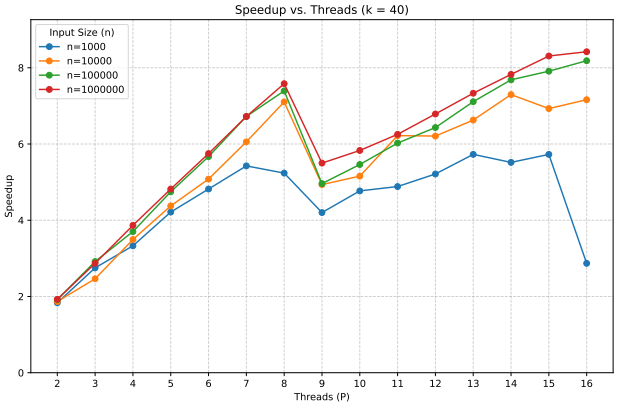


Figure 5: Speedup vs. Threads, k=40

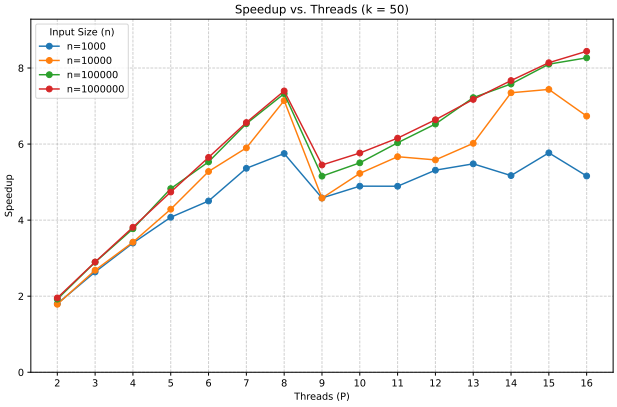


Figure 6: Speedup vs. Threads, k=50

6. Conclusions

This report has presented a comprehensive study of the kmeans clustering algorithm, from its theoretical foundations to practical implementation and performance evaluation in both sequential and parallel forms.

We began by formally introducing the mathematical formulation of kmeans clustering, describing its objective function, iterative procedure, and time complexity. This theoretical background established a solid foundation for understanding the computational characteristics of the algorithm and the motivation for parallelization.

Following this, we detailed the design and implementation of a sequential baseline, ensuring that all performance comparisons were made against an efficient and optimized reference point. We then introduced a fully parallel version of the algorithm built using OpenMP, which exploited the data-parallel nature of k-means. Particular attention was given to memory layout, thread-local data structures, synchronization strategies, and the use of OpenMP constructs such as `nowait`, `parallel for`, and `critical` to ensure correctness and scalability.

To rigorously evaluate performance, we implemented a flexible and reproducible benchmarking framework capable of generating synthetic datasets, running multiple trials for statistical stability, and measuring speedup across a variety of configurations. We then conducted extensive experiments, varying the dataset cardinality n for different, fixed values of the cluster count k, and measured speedup as a function of thread count.

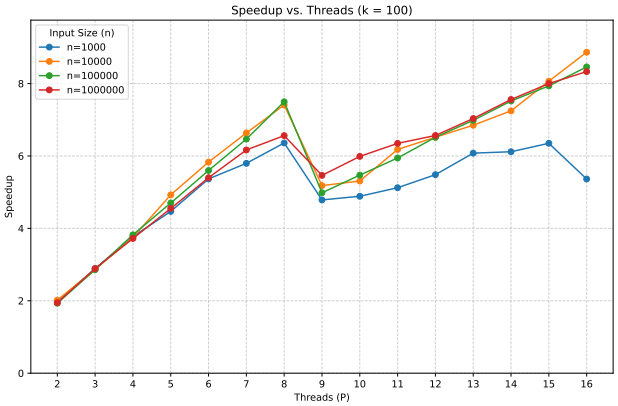


Figure 7: Speedup vs. Threads, k=100

The performance evaluation yielded several insights into the scalability of the parallel algorithm. The speedup was shown to scale nearly linearly with the number of physical cores, consistently peaking at 8 threads, which directly corresponds to the hardware architecture of the test system. The transition to logical cores via Hyper-Threading, beginning at 9 threads, resulted in a performance dip followed by more modest gains, ultimately achieving a maximum speedup of approximately 8.5x. In conclusion, this study demonstrates that OpenMP-based parallelization is an effective approach to accelerating k-means clustering on shared-memory architectures.