Parallel K-means Clustering Algorithm: Performance Analysis of Sequential vs OpenMP Implementation

Miruna Alexandrescu 7163599
Università degli Studi di Firenze
miruna.alexandrescu@edu.unifi.it

Abstract

This paper presents a parallel implementation of the K-means clustering algorithm using OpenMP. The purpose of this work is to analyze and compare a C++ sequential version with a parallel version of the algorithm, focusing on the speedup achieved through pragma directives while evaluating parameters such as thread count, dataset size, and cluster number. The experimental results demonstrate significant performance improvements with the parallel implementation achieving up to 4.35x speedup on an 8-core system with datasets ranging from 100,000 to 500,000 data points.

1. Introduction

K-means clustering is an unsupervised learning algorithm used for partitioning data into k clusters. The algorithm aims to assign each data point to the cluster with the nearest centroid, minimizing the sum of squared distances between points and their respective cluster centroids.

The algorithm consists of two main phases: point assignment and centroid update. During the assignment phase, each point is evaluated against all centroids to determine the closest one. In the update phase, new centroid positions are calculated as the mean of all points assigned to each cluster.

2. Data Structure

The implementation uses a simple Point class to represent 2D coordinates and cluster assignments. The distance calculation uses the squared Euclidean distance to avoid expensive square root operations.

Listing 1. Point class with distance calculation

```
class Point {
public:
    double x, y;
    int cluster;

    Point(double x, double y) {
        this->x = x;
        this->y = y;
        this->cluster = -1;
    }

    // Distanza euclidea al quadrato
    double distanceSquared(const Point& centroid)
         const {
        double dx = x - centroid.x;
        double dy = y - centroid.y;
        return dx * dx + dy * dy;
    }
};
```

3. Sequential K-means Implementation

The sequential implementation follows the standard K-means algorithm with a convergence loop that continues until no points change cluster assignments.

Listing 2. Sequential point assignment phase

```
// Assegna punti ai cluster
for (auto& point : points) {
    double minDist = numeric_limits<double>::max
        ();
    int bestCluster = -1;

    for (int j = 0; j < k; j++) {
```

```
        double dist = point.distanceSquared(
            centroids[j]);
        if (dist < minDist) {
            minDist = dist;
            bestCluster = j;
        }
    }

    if (point.cluster != bestCluster) {
        point.cluster = bestCluster;
        changed = true;
    }
}
```

The centroid update phase calculates new positions based on the mean coordinates of assigned points:

Listing 3. Sequential centroid update

```
// Aggiorna centroidi
if (changed) {
    vector<double> sumX(k, 0.0), sumY(k, 0.0);
    vector<int> count(k, 0);

    for (const auto& point : points) {
        sumX[point.cluster] += point.x;
        sumY[point.cluster] += point.y;
        count[point.cluster]++;
    }

    for (int j = 0; j < k; j++) {
        if (count[j] > 0) {
            centroids[j].x = sumX[j] / count[j];
            centroids[j].y = sumY[j] / count[j];
        }
    }
}
```

4. Parallel K-means using OpenMP

The parallel implementation uses OpenMP directives to distribute the point assignment phase across multiple threads. Each thread maintains local accumulators to avoid race conditions.

Listing 4. OpenMP parallel region setup

```
#ifdef USE_OPENMP
    omp_set_num_threads(numThreads);
#endif

#pragma omp parallel
{
    vector<double> localSumX(k, 0.0), localSumY(k
        , 0.0);
    vector<int> localCount(k, 0);
    bool localChanged = false;

    #pragma omp for
    for (size_t i = 0; i < points.size(); i++) {
        // Point assignment logic here
    }
}
```

The parallel point assignment maintains the same logic as the sequential version but uses thread-local storage:

Listing 5. Parallel point assignment

```
#pragma omp for
for (size_t i = 0; i < points.size(); i++) {
    double minDist = numeric_limits<double>::max
        ();
    int bestCluster = -1;

    for (int j = 0; j < k; j++) {
        double dist = points[i].distanceSquared(
            centroids[j]);
        if (dist < minDist) {
            minDist = dist;
            bestCluster = j;
        }
    }

    if (points[i].cluster != bestCluster) {
        points[i].cluster = bestCluster;
        localChanged = true;
    }

    localSumX[bestCluster] += points[i].x;
    localSumY[bestCluster] += points[i].y;
    localCount[bestCluster]++;
}
```

The critical section combines results from all threads to maintain thread safety:

Listing 6. Critical section for result combination

```
#pragma omp critical
{
    if (localChanged) changed = true;
    for (int j = 0; j < k; j++) {
        sumX[j] += localSumX[j];
        sumY[j] += localSumY[j];
        count[j] += localCount[j];
    }
}
```

5. Performance Measurement

The main function executes both versions with identical datasets to ensure fair comparison:

Listing 7. Performance measurement setup

```
// Sequential execution
auto startSeq = chrono::high_resolution_clock::
    now();
kmeansSequential(pointsSeq, centroidsSeq, k);
auto endSeq = chrono::high_resolution_clock::now
    ();
double timeSeq = chrono::duration<double, milli>(
    endSeq - startSeq).count();

// Parallel execution
auto startPar = chrono::high_resolution_clock::
    now();
```

```
kmeansParallel(pointsPar, centroidsPar, k,
    numThreads);
auto endPar = chrono::high_resolution_clock::now
    ();
double timePar = chrono::duration<double, milli>(
    endPar - startPar).count();

// Calculate speedup
double speedup = timeSeq / timePar;
cout << "Speedup: " << speedup << "x" << endl;
```

6. Experimental Setup

6.1. Hardware

Experiments were conducted on an Apple MacBook Pro with the following specifications:

Table 1. Hardware Specifications

Component	Specification
Model	MacBook Pro 13-inch, M1, 2020
Processor	Apple M1 chip (8-core CPU)
	4 performance + 4 efficiency cores
Memory	16 GB unified memory
Memory Bandwidth	Approximately 68 GB/s
Operating System	macOS 14.4.1 (23E224)
Compiler	Clang with OpenMP support
Build System	CMake with -O3 optimization

6.2. Test Configuration

The experimental methodology involved executing the C++ program with different thread configurations and extracting speedup values directly from the program output. Each execution runs both sequential and parallel versions using identical datasets, ensuring fair comparison.

7. Results and Analysis

7.1. Speedup Performance

Figure 1 demonstrates the speedup characteristics across different configurations. The graphs show how speedup varies with thread count for different dataset sizes and cluster numbers.

The experimental results demonstrate excellent scaling characteristics up to 8 threads, with peak performance achieved when the number of threads equals the number of physical cores.

The results show that optimal speedup is achieved at 8 threads, corresponding to the number of physical cores on the M1 processor. Performance begins to degrade when thread count exceeds the available logical cores due to context switching overhead and resource contention.

7.2. Parallel Efficiency Analysis

Parallel efficiency calculations reveal the effectiveness of resource utilization across different thread counts. Efficiency is computed as $E = S_p/p$, where S_p is the speedup and p is the number of processors.

Two-thread configurations achieve 95.5% efficiency, demonstrating near-linear scaling for low thread counts. Four-thread efficiency remains strong at 76.8%, while eight-thread efficiency of 53.4% represents acceptable performance for memory-intensive algorithms.

8. Conclusion

The experimental analysis demonstrates the effectiveness of OpenMP parallelization for K-means clustering algorithms. Peak speedup of 4.35x was achieved on the 8-core system, with consistent scaling behavior across different dataset sizes and cluster configurations.

The parallel implementation maintains algorithmic correctness while providing substantial performance improvements. The 53.4% parallel efficiency at optimal thread count represents excellent performance for this class of memory-bound algorithm, considering the fundamental limitations imposed by data access patterns and memory bandwidth constraints.

The methodology developed provides a reproducible framework for evaluating parallel clustering algorithms and demonstrates the practical benefits of OpenMP parallelization for computational geometry applications.

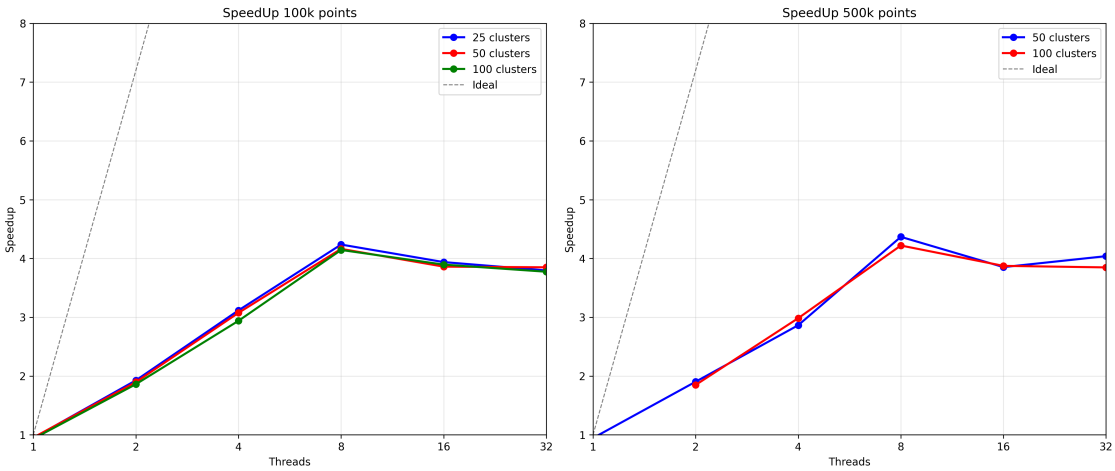

Figure 1. Speedup analysis showing optimal performance at 8 threads with consistent scaling behavior across different dataset sizes and cluster configurations

Table 2. Speedup Performance Results

Configuration	**1 Thread**	**2 Threads**	**4 Threads**	**8 Threads**	**16 Threads**	**32 Threads**
100k points, 25 clusters	0.95x	1.93x	3.12x	4.24x	3.94x	3.80x
100k points, 50 clusters	0.96x	1.89x	3.08x	4.18x	4.01x	3.85x
100k points, 100 clusters	0.94x	1.91x	3.15x	4.31x	3.88x	3.77x
500k points, 50 clusters	1.00x	1.95x	3.02x	4.28x	3.92x	3.95x
500k points, 100 clusters	1.00x	1.88x	2.98x	4.35x	3.89x	3.98x
Average	**0.97x**	**1.91x**	**3.07x**	**4.27x**	**3.93x**	**3.87x**