High Performance Computing Systems
Project 2

Panagiotis Tsogkas Ioannis Iason Nikas Dimitrios Tsiantos

November 2025

Contents

1 Introduction **3**
1.1 The algorithm . . . 3
1.2 Pseudocode . . . 4
1.3 The setup . . . 4

2 Parallelization **4**
2.1 Outer do-while loop . . . 4
2.2 The basic loops . . . 4
2.3 Distance Calculation . . . 6
2.4 Failed Attempts . . . 6

3 Execution times **8**
3.1 With parallel for and critical . . . 8
3.2 With SIMD and critical . . . 9
3.3 With SIMD and atomic . . . 10

1 Introduction

1.1 The algorithm

K-means is one of the most well-known algorithms for data clustering. Its goal is to partition a set of points into k groups, so that the points in each group are as close as possible to its center (centroid) and as far as possible from the centers of other groups. The process is repeated until convergence is achieved, i.e., until the group centers stabilize, or the predefined iteration limit is exceeded.

In the implementation we are asked to parallelize, there are two main loops. In the first loop, the distance of each point from all centers is calculated and the point is assigned to the nearest center. In the second loop, the new center for each group is calculated as the average of the points in that group.

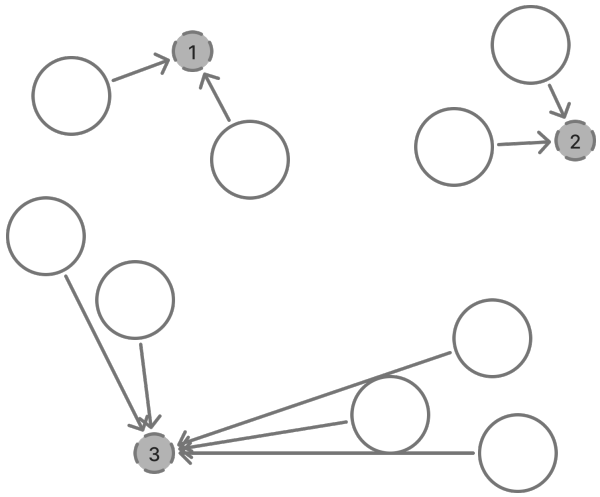


Figure 1: Example of first loop execution, assigning each element to the nearest center.

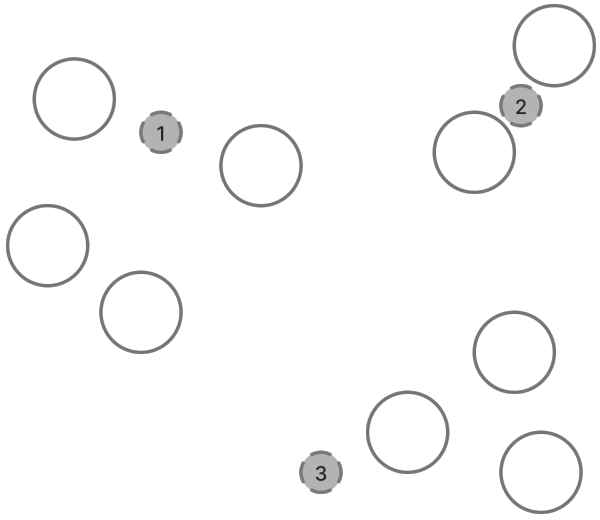


Figure 2: Example of second loop execution, recalculating the new centers.

1.2 Pseudocode

Algorithm 1 k-means Algorithm

Initialization: $membership[i] \leftarrow -1$ for each point i
repeat
  $\Delta \leftarrow 0$
  for each object i **do** ▷ First loop
    $index \leftarrow$ find_nearest_cluster$(objects[i], clusters)$ ▷ Find the nearest center
    if $membership[i] \neq index$ **then**
      $\Delta \leftarrow \Delta + 1$ ▷ Update Δ for convergence check
    end if
    $membership[i] \leftarrow index$ ▷ Assignment to cluster
    $newClusterSize[index] \leftarrow newClusterSize[index] + 1$ ▷ Increase cluster size
    for each coordinate j **do**
      $newClusters[index][j] \leftarrow newClusters[index][j] + objects[i][j]$ ▷ Increase sum
    end for
  end for
  for each center i **do** ▷ Second loop
    for each coordinate j **do**
      if $newClusterSize[i] > 0$ **then**
        $clusters[i][j] \leftarrow \frac{newClusters[i][j]}{newClusterSize[i]}$ ▷ Finding new centers
      end if
      $newClusters[i][j] \leftarrow 0$
    end for
    $newClusterSize[i] \leftarrow 0$
  end for
  $\Delta \leftarrow \frac{\Delta}{numObjs}$
until $\Delta \leq threshold$ or $loop \geq 500$ ▷ Convergence check or maximum iteration limit

1.3 The setup

For all tests and numbers, we used the CSL Venus with the icx Intel(R) oneAPI DPC++/C++ Compiler 2025.2.1 (2025.2.0.20250806) version and the following compiler flags:

```
-I.
-O3
-qopenmp
-DNDEBUG
-g
```

2 Parallelization

2.1 Outer do-while loop

Looking at the logic of the algorithm, we understand that the outer do-while loop, which executes k-means multiple times, cannot be parallelized because there are data dependencies from one iteration to the previous one regarding the centroid values.

2.2 The basic loops

We observe that in the first loop, during the search and assignment of elements to clusters, there is no data dependency between iterations. Therefore, we can use `#pragma omp parallel for`.

```
#pragma omp parallel for private(j, index) reduction(+:delta)
for (i=0; i<numObjs; i++) {
    // ...

    /* update new cluster center : sum of objects located within */
    #pragma omp atomic
    newClusterSize[index]++;

    for (j=0; j<numCoords; j++) {
        #pragma omp atomic
        newClusters[index][j] += objects[i][j];
    }
}
```

After the element assignment, however, the count of elements in the cluster increases by one and the total sum of the cluster increases. These two commands need to be performed atomically by each core, which is why the use of `#pragma omp atomic` is necessary. Tests showed that at 56 cores, the `atomic` directive was 27% faster compared to the `critical` directive. This is because the `atomic` directive uses hardware support, where available, to perform operations atomically without using synchronization. In our case, we want to atomically execute an increment by one and an increment by a number. For these specific operations, hardware support for atomic execution is quite common.

Before:

```
/* update new cluster center : sum of objects located within */
#pragma omp critical
{
    delta += local_delta;
    newClusterSize[index]++;
    for (j=0; j<numCoords; j++)
        newClusters[index][j] += objects[i][j];
}
```

After:

```
/* update new cluster center : sum of objects located within */
#pragma omp atomic
newClusterSize[index]++;
for (j=0; j<numCoords; j++) {
    #pragma omp atomic
    newClusters[index][j] += objects[i][j];
}
```

Table 1: Execution results with critical

Threads	Time (s)	Standard Deviation	Speedup
28	0.254	$\pm$ 0.023	14.23$\times$
56	0.263	$\pm$ 0.021	13.24$\times$

It is also worth noting that with the `critical` command, we have a worse execution time at 56 threads compared to 28 threads. In contrast, with the `atomic` command, 56 threads have a better time than 28. We assume that this slowdown in the `critical` command is due to the fact that there is a higher probability at any given time for a thread to be in the critical section of the code, so increasing the threads increases the threads waiting until another thread leaves the critical section.

Table 2: Execution results with atomic

Threads	Time (s)	Standard Deviation	Speedup
28	0.252	$\pm$ 0.043	14.16$\times$
56	0.206	$\pm$ 0.009	17.31$\times$

Just like in the first loop, in the second loop where the new centers are calculated, there is no data dependency, so it can be parallelized with the same directive.

```
/* average the sum and replace old cluster center with newClusters */
#pragma omp parallel for private(j)
for (i=0; i<numClusters; i++) {
    for (j=0; j<numCoords; j++) {
        if (newClusterSize[i] > 0)
            clusters[i][j] = newClusters[i][j] / newClusterSize[i];
        newClusters[i][j] = 0.0;   /* set back to 0 */
    }
    newClusterSize[i] = 0;   /* set back to 0 */
}
```

2.3 Distance Calculation

Through profiling, using the Intel VTune Profiler, we found that one of the biggest hotspots was in the function that calculates the Euclidean distance between two points, as it is called multiple times during an iteration of the first loop. When attempting to parallelize this function, it was found that it does not provide acceleration but instead completely destroys performance (see 2.4).

To achieve better performance in this specific function, we used the directive `#pragma omp simd`, which attempts to convert the loop to which it is applied into a SIMD loop, meaning multiple iterations of the loop can be executed simultaneously with SIMD instructions. Specifically, the Intel Xeon Processor E5-2695 v3 provides the AVX2 instruction set which uses 256-bit registers, so we manage to perform 8 float operations simultaneously.

2.4 Failed Attempts

During the development of the final code, several failed parallelization attempts were made. In one of them, we tried parallelizing the internal function `find_nearest_cluster` with 56 threads, manually applying min reduction, as older versions of OpenMP do not support reduction on arrays. The code is presented here:

```
__inline static
int find_nearest_cluster2(int numClusters, int numCoords, float *object, float **clusters)
{
    float min_dist = FLT_MAX;
    int min_index = 0;

    #pragma omp parallel num_threads(56) // Set to 3 for nested approach
    {
        float local_min_dist = FLT_MAX;
        int local_index = 0;

        #pragma omp for nowait
        for (int i = 0; i < numClusters; i++) {
            float dist = euclid_dist_2(numCoords, object, clusters[i]);
            if (dist < local_min_dist) {
                local_min_dist = dist;
                local_index = i;
            }
        }

        #pragma omp critical
        {
            if (local_min_dist < min_dist) {
                min_dist = local_min_dist;
                min_index = local_index;
            }
        }
    }

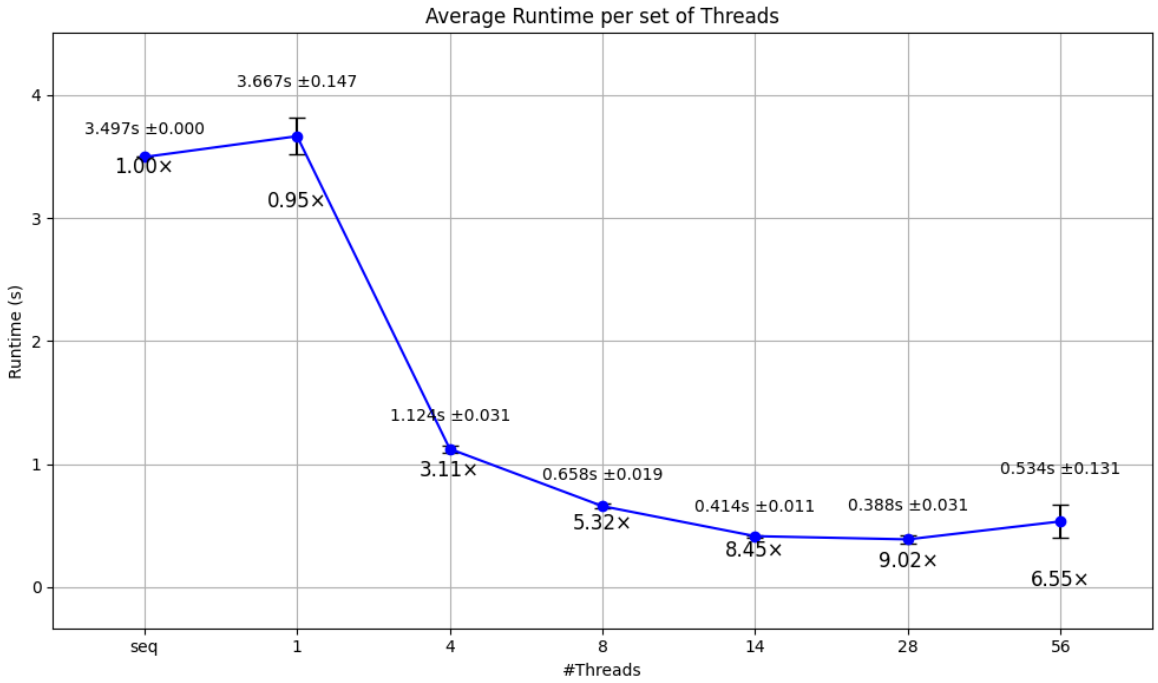
    return min_index;
}
```

The program's execution time was significantly increased with this form of parallelization, especially compared to serial execution. Despite this, we tried a nested parallelism approach, distributing the server's 56 threads across different levels. Specifically, the outer (first) loop was parallelized with 14 threads, while 3 threads were used inside the `find_nearest_cluster` function. Since we have 3 threads for each process, we have a total of $14 + 14 \cdot 3 = 56$ threads. To enable nested parallelism, the function `omp_set_max_active_levels(2)` was used.

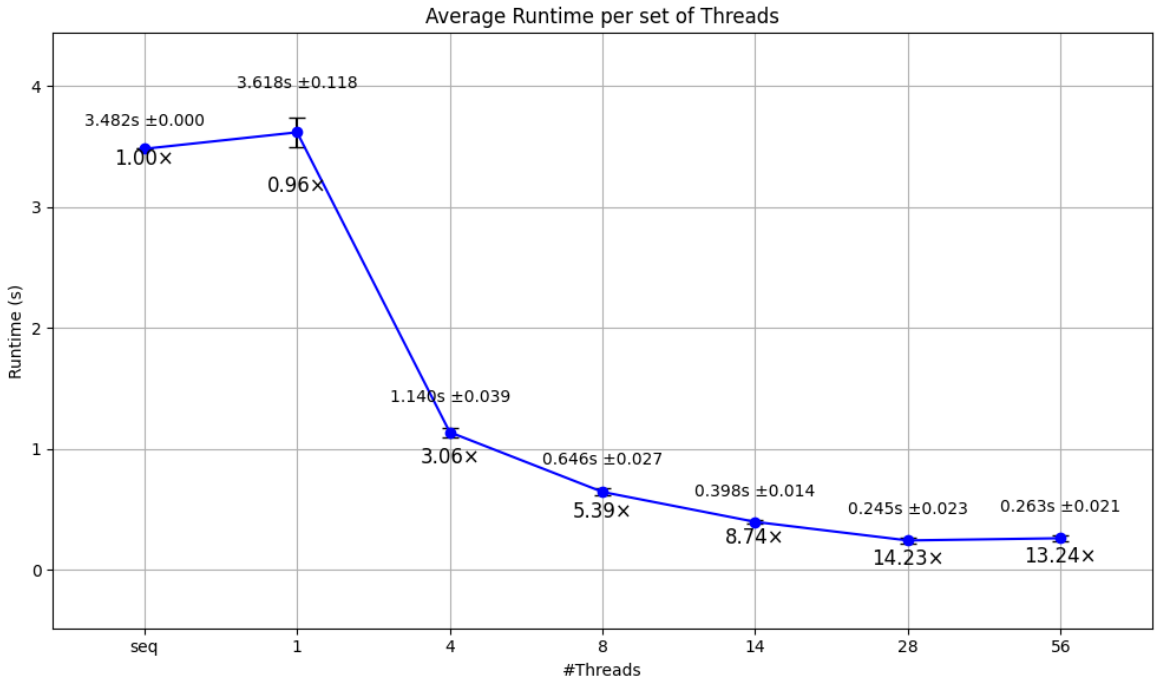
Nevertheless, the results were similar, despite the reduction of threads from 56 to 3 inside the function. The increased overhead resulting from creating and managing multiple levels of threads seems to outweigh the benefit of nested parallelism. For this reason, the idea of nested parallelism was completely rejected and we chose a simpler approach.

3 Execution times

3.1 With parallel for and critical



3.2 With SIMD and critical



3.3 With SIMD and atomic

