PC-2023/24 Parallel K-Means

Author
E-mail address
francesco.marchini1@edu.unifi.it

Abstract

This report delves into the utilization of OpenMP as an API for parallelizing C++ code, with a specific emphasis on the k-means algorithm employing Euclidean distance. The analysis compares a sequential version with a parallel version of the algorithm, focusing on the efficiency of parallelization achieved through OpenMP pragma directives. The report also provides insights into the parallelization process, highlighting the key areas of focus and the utilization of OpenMP directives.

Future Distribution Permission

1. Introduction

This report describes the use of *OpenMP* [1] as a API for parallelizing *C++* code. We will analyse and compare two implementations of the k-means algorithm with Euclidean distance: a sequential version and a parallel version. We will focus on the parallelization efficiency of our *OpenMP* pragma directive code, going to estimate the speedup achieved in terms of execution time. The algorithm analyses *1,000,000* data points ranging from 0 to *1,000* in an attempt to identify *10* clusters. The code repository is available on *GitHub* by clicking on the following link: parallel-k-means.

2. K-Means algorithm

This section explains how the code works, focusing on the most relevant parts for the future parallelization. The implementation of the k-means algorithm can be divided into two main parts:

- setup and data generating process;
- clusters finding and points association.

The initial part of the code is not parallelizable as it is solely responsible for setting up the environment for subsequent operations. On the other hand, the second part is of our interest; this is where we are going to act with `pragma` directives to make the code multi-threaded.

2.1. Parallelization

The code's second portion is divided into two main operations:

- assignment of each point to its nearest centroid;
- calculate the average distance between the points of each centroid and assign the new centroid.

2.1.1 First section

The first operation consist on calculate the Euclidean distance between points and all clusters, assigning all points to the nearest cluster. This operation is implemented through the use of two for loops, the first dealing with scrolling through all the points in the randomly generated dataset, the second scrolling through all the clusters by going to associate the point in question with the nearest cluster.

```
for (int i = 0; i < points.size(); i++)
{
    double min_distance = INFINITY;
    int centroid_index = -1;
    for (int j = 0; j < centroids.size();
        j++)
    {
        double distance =
            euclidean_distance(points[i],
            centroids[j]);
        if (distance < min_distance)
        {
            min_distance = distance;
            centroid_index = j;
        }
    }
    if (centroid_index !=
        points[i].clusterId)
    {
        points[i].clusterId =
            centroid_index;
        changed = true;
    }
}
```

Focusing on the external for loop, we can easily see that the operation of calculating the nearest cluster for each point is independent of each other, so we can simply parallelise the loop using the *OpenMP* directive, paying attention to the variables shared by the various threads: `#pragma omp parallel for shared (points, centroids, cumulate_centroids, changed)`.
In addition, we must make the operations on newly declared shared variables atomic, to avoid race condition. This can be achieved by adding the `#pragma omp atomic write` directive.

```
#pragma omp parallel for shared(points,
    centroids, cumulate_centroids, changed)
for (int i = 0; i < points.size(); i++)
{
    double min_distance = INFINITY;
    int centroid_index = -1;
    for (int j = 0; j < centroids.size();
        j++)
    {
        double distance =
            euclidean_distance(points[i],
            centroids[j]);
        if (distance < min_distance)
        {
            min_distance = distance;
            centroid_index = j;
        }
    }
    if (centroid_index !=
        points[i].clusterId)
    {
        #pragma omp atomic write
        points[i].clusterId =
            centroid_index;
        #pragma omp atomic write
        changed = true;
    }
}
```

2.1.2 Second section

The second operation involves computing the average distance between the internal points of each cluster. In this case, a for loop is used to achieve this goal. It scrolls through all clusters and calculates the mean distance between all points in the cluster, assigning the new value to them.

```
for (int i = 0; i < centroids.size(); i++)
{
    centroids[i].x =
        average_distance(cumulate_centroids[i].x);
    centroids[i].y =
        average_distance(cumulate_centroids[i].y);
    centroids[i].z =
        average_distance(cumulate_centroids[i].z);
}
```

All operations within the loop are independent of each other. Therefore, as before, we can parallelise the cycle. The 'for' loop can be parallelized by adding the `#pragma omp parallel for shared(centroids, cumulate_centroids)` directive. Here is the final code:

```
#pragma omp parallel for shared(centroids,
    cumulate_centroids)
for (int i = 0; i < centroids.size(); i++)
{
    centroids[i].x =
        average_distance(cumulate_centroids[i].x);
    centroids[i].y =
        average_distance(cumulate_centroids[i].y);
    centroids[i].z =
        average_distance(cumulate_centroids[i].z);
}
```

3. Speedup

The performance achieved with the sequential version will now be compared to that of the parallel version in terms of execution time. In order

to reach this task, a Python script will be utilized to execute both sequential and parallel versions of k-means varying the number of threads used. The script calculates the speedup achieved at each iteration and generates a graph displaying the results in terms of speedup for threads. All tests were performed on an 11th generation Intel i7 [2] CPU with 4 cores and 8 threads.
The speed up is computed as follow:

$$S_P = t_s/t_p \tag{1}$$

Where t_s is the execution time of sequential program and t_p the execution time of the parallel program. In this case, we run the parallel code using between one and sixteen threads to evaluate performance at each step. As expected given the CPU used, the best approach is achieved by utilising eight threads: $S_P = 2.5076$. All result are available in Fig. 1. Considering the 8 threads of the 11th Intel i7, we will measure our speedup in terms of efficiency using the following formula:

$$E_P = S_P/P \tag{2}$$

Where P is number of CPU units used. Unfortunately, the achieved efficiency is not very high: $E_P = 0.313$; highlighting an efficiency of only 31%.
The second graph (Fig. 2) compares the sequential and parallel programs based on their execution time in 20 random trials, each with a different dataset. The x-axis shows the iterations required for the algorithm to converge, and the y-axis shows the execution time in seconds. Finally, the last experiment conducted concerns the comparison of the execution times of only the parallel code when varying the number of threads used. Refer to the graph in Figure 3 to visualise the results.

4. Conclusion

We can conclude that by using OpenMP, we can achieve a good degree of parallelism with just a few `pragma` instructions. We recommend experimenting with further tests on different machines with diffirent numbers of CPUs and threads, highlighting any differences in speedup achieved.

References

[1] Openmp, 2012. https://www.openmp.org/.
[2] Intel. i7 11th generation, 2021. https://www.intel.com/.

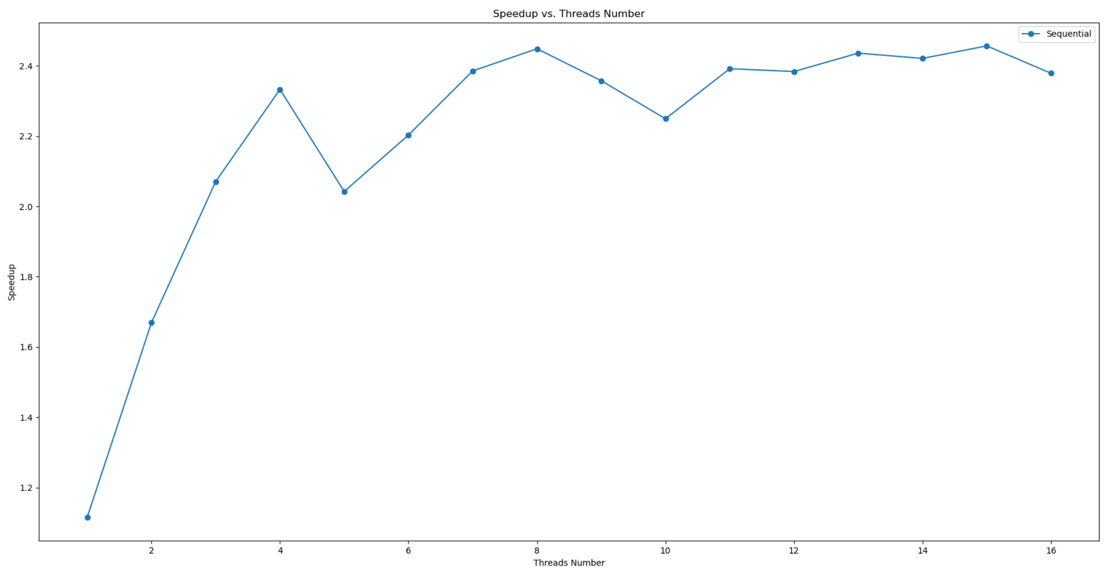

Figure 1. Num of threads vs Speedup

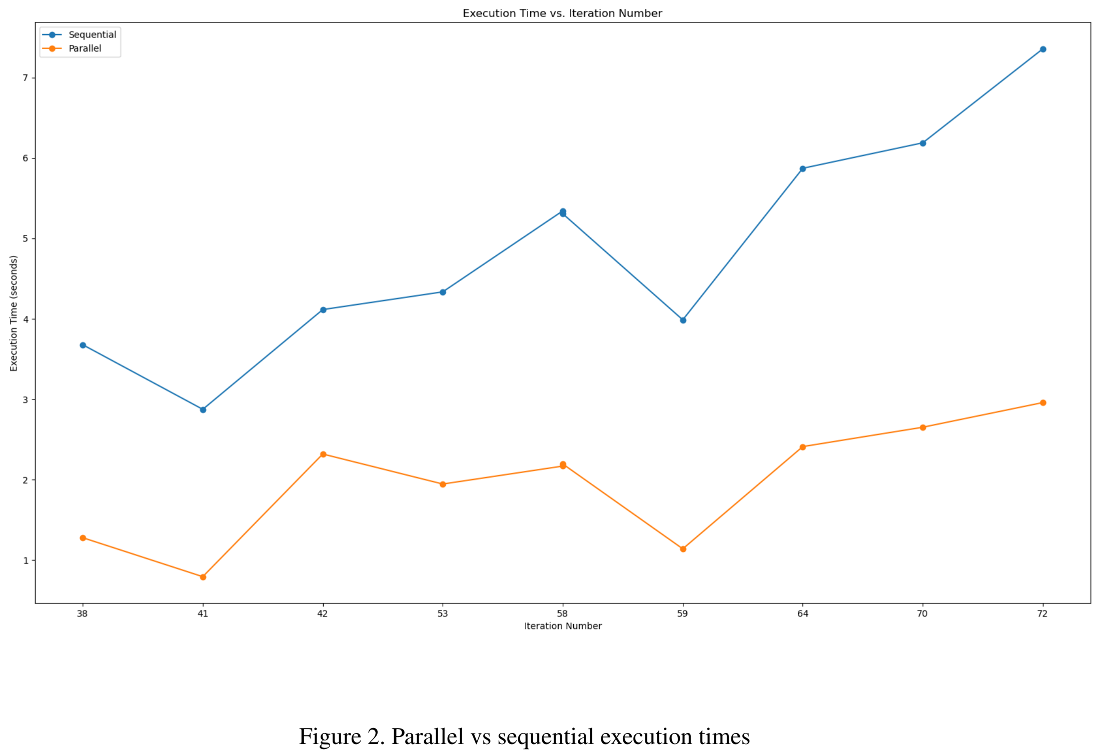

Figure 2. Parallel vs sequential execution times

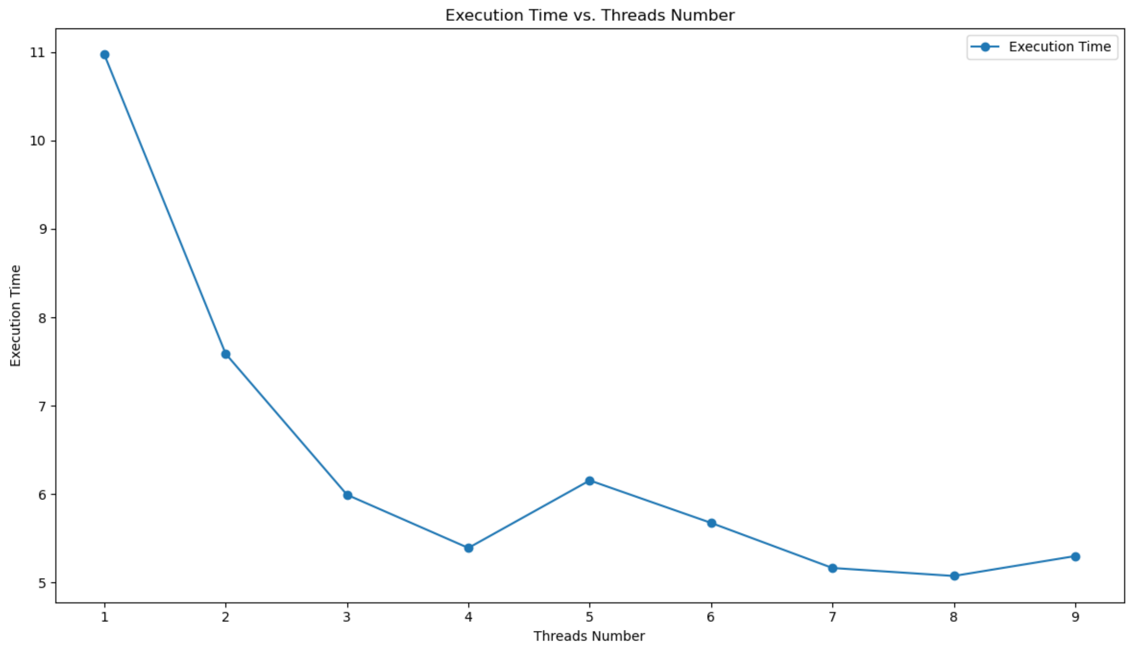

Figure 3. Execution time VS Number of Threads