LAB 2

Christodoulos Zerdalis 03531 Charalampos Malakoudis 03516

November 3, 2025

1 Introduction

The objective of this lab is to parallelize the k-means clustering algorithm for CPU execution. We implemented the parallelization using the OpenMP library, supplemented by traditional synchronization methods and changing the code architecture in order to prevent race conditions between threads. The final code was then tested on the csl-venus machine using 1, 4, 8, 14, 28 and 56 threads to evaluate its performance. Through testing we also picked 36 as the number of chunks used in the scheduling of our parallel for loops.

2 Optimizations

Since the objective is to parallelize the k-means algorithm using the OpenMp library we focused on parallelizing all the available loops and testing which of them made the biggest impact in terms of performance. Most of the algorithm's computational complexity is located in the do-while-loop of the seq_kmeans function. Inside it there are two for-loops, the first one is responsible for finding the nearest cluster of every object using the find_nearest_cluster function and the second to calculate the position of the new clusters. The first loop is more computationally intense than the second, since the find_nearest_cluster function is essentially made up of two nested for-loops making the whole thing a triple nested for-loop. Since we want to truly parallelize the process, meaning that every thread created must run on the CPU, we have to take into account that the number of real threads on a CPU are limited. This means that in the case of nested loops we can only parallelize one. After testing we decided that parallelizing the outer loops is the best option.

2.1 First Loop

In order to parallelize the first loop, we used ‘omp for‘ inside a parallel region, privatizing the variables i, j, and index. We also handled the delta += 1 calculation using the reduction clause. Afterwards, we had to address race conditions when writing to the newclustersize and newClusters arrays by making these lines of code atomic. Making these lines atomic is essential for correctness but harms performance, since only one thread at a time can execute this part of the code. Also, in the loop inside the euclid_dist_2 function, we used ‘omp simd reduction‘ to make the calculation of the ans variable faster.

2.2 Second Loop

For the second loop we just used ‘omp for‘ while privatizing the i and j iterators.

2.3 Algorithmic improvements

The biggest weakness of the algorithm is the race conditions which usually require the ‘atomic‘ flag in order to fix. This theoretically slows down the execution time. One way to solve it is to

create thread-unique arrays so that we don't have data overlapping issues. To implement that we need to allocate a lot of extra memory and add one extra for-loop that combines the results stored in each private array into the main one. This sequence is also parallelized using 'omp for' and privatizing the iterators. While this solution seemed to work in theory ,as it removes the race conditions, the extra time needed for the new for-loops results in almost identical times. Regardless, we use this version for our tests as it is theoretically better and we are more proud of it. Lastly, we parallelized the initialization of the dynamic arrays which made a subtle difference.

3 Schedule formats

For both loops we started by testing the use of static scheduling which made more sense since the workload for every loop seemed the same. This worked best for the second loop but for the first loop the best scheduling policy is the dynamic. In our opinion this happens because the find_nearest_cluster function has varying runtimes and alters the time needed for each loop to finish. In order to further optimize our scheduling we tested the program with 14 and 56 threads and varied the chunk sizes in the range of 1, 4, 8, 16, 32, 128, 512, 1000 . According to the results, we deduced that the best chunk number is 32. We can view the numbers in the following chart.

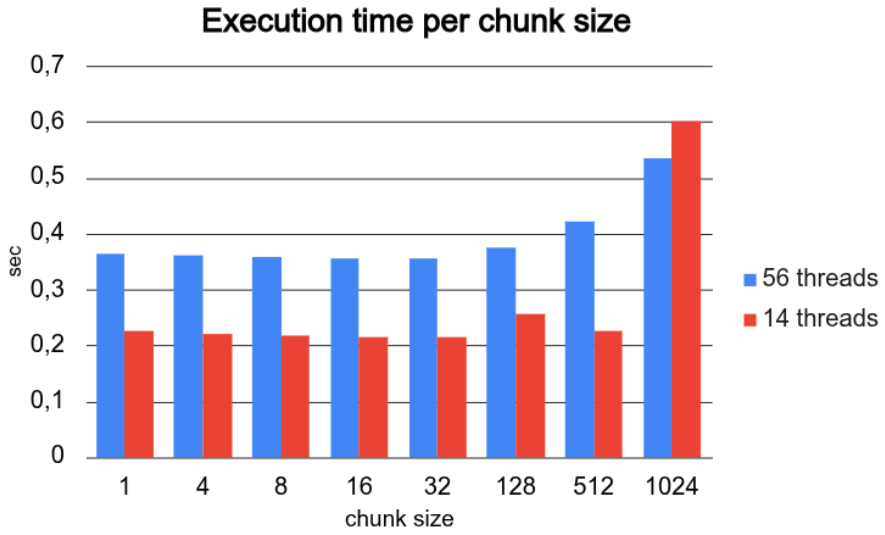

Figure 1: Here we can see that the chunk sizes of 16 and 32 have the best performance

Chunks	1	4	8	16	32	128	512	1024
14 threads	0.363515	0.360665	0.357585	0.356495	0.356965	0.373575	0.420830	0.536035
56 threads	0.225245	0.219945	0.216565	0.216040	0.215205	0.256435	0.225245	0.600695

4 Results

In order to view the results of our improvements in the runtime of the program, we constructed the following charts, showing the reduction in runtime ,in comparison to the runtime of the original sequential version.
What is interesting in the graph and in the numbers is that the program performs slightly better with 28 threads than with 56! That could be attributed to the fact that we are running the program on the csl-venus which essentially has 2 cpus with 14 physical cores each, with the ability to run 56 logical threads through hyperthreading. When using more that 28 threads the

program is running on two separate cpus, this can cause delay since the inter communication of the two cpus is very slow compared to the intra communication of the cores in each. Also the program might just suffer from delay from the creation and scheduling of the threads when it runs on 56 of them.
We also made a chart with the theoretical speedup that supplying more threads would give to the runtime of the program in relation to the actual speedup we experienced. The chart clearly shows that the program does not speed up perfectly in relation to the threads used and experiences a fraction of the runtime gain that was theoretically guessed.
Lastly, All our tests were made by compiling the program with the flags -qopenmp, icx (instead of gcc) and -O3 (instead of -O4).

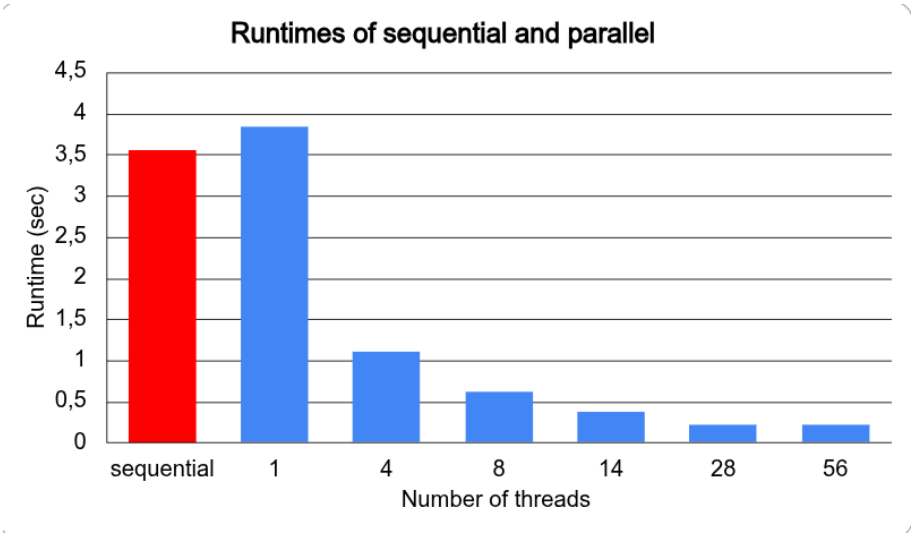

Figure 2: Here we can see the declining improvement of extra threads

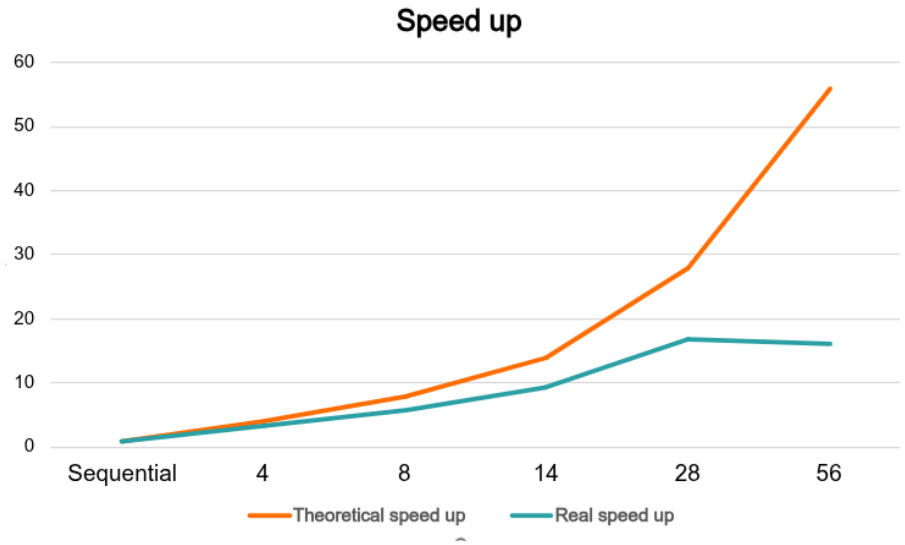

Figure 3: Here we can see the difference between theoretical and real speedup