Lab 2

High Performance Computing
Fall semester 2025-2026

Aikaterini Tsiaousi 3626
Panagiotis Nanousis 3599

November 1, 2025

1 Computer Hardware used

The computer that was used to generate the execution times had the below characteristics:

1. **CPU**: Ryzen 9 7900x3D
2. **Memory**: 2x 16GB Corsair Vengeance DDR5 @54000MHZ
3. **Operating System**: Omarchy v3.0.2
4. **Kernel**: Linux 6.17.1-arch1-1
5. **Compiler**: gcc (GCC) 15.2.1 20250813 / Intel(R) oneAPI DPC++/C++ Compiler 2025.2.1

2 Methodology

The exercise asks us to improve the execution time of the k-means algorithm by parallelizing it with the Open MP library. To start, we profiled the sequential code in order to find the bottlenecks of the code. After that, we parallelized the bottlenecks in order to gain performance in comparison with the original code. K-means is a massively parallel problem, meaning it can be parallelized to its theoretical P speedup where P is the number of processors utilized.

3 Profilling

In order to profile the code, we measured the time taken in different sections of the sequential code that we thought would be the main bottlenecks. The code had two logical sections, the initialization and the computation ones that were separated into one for loop and one do-while loop. The first for loop is pretty straight forward so we measure the time it takes on its own. The second loop, which does that computations, is more complex, hence we split it into further sections.

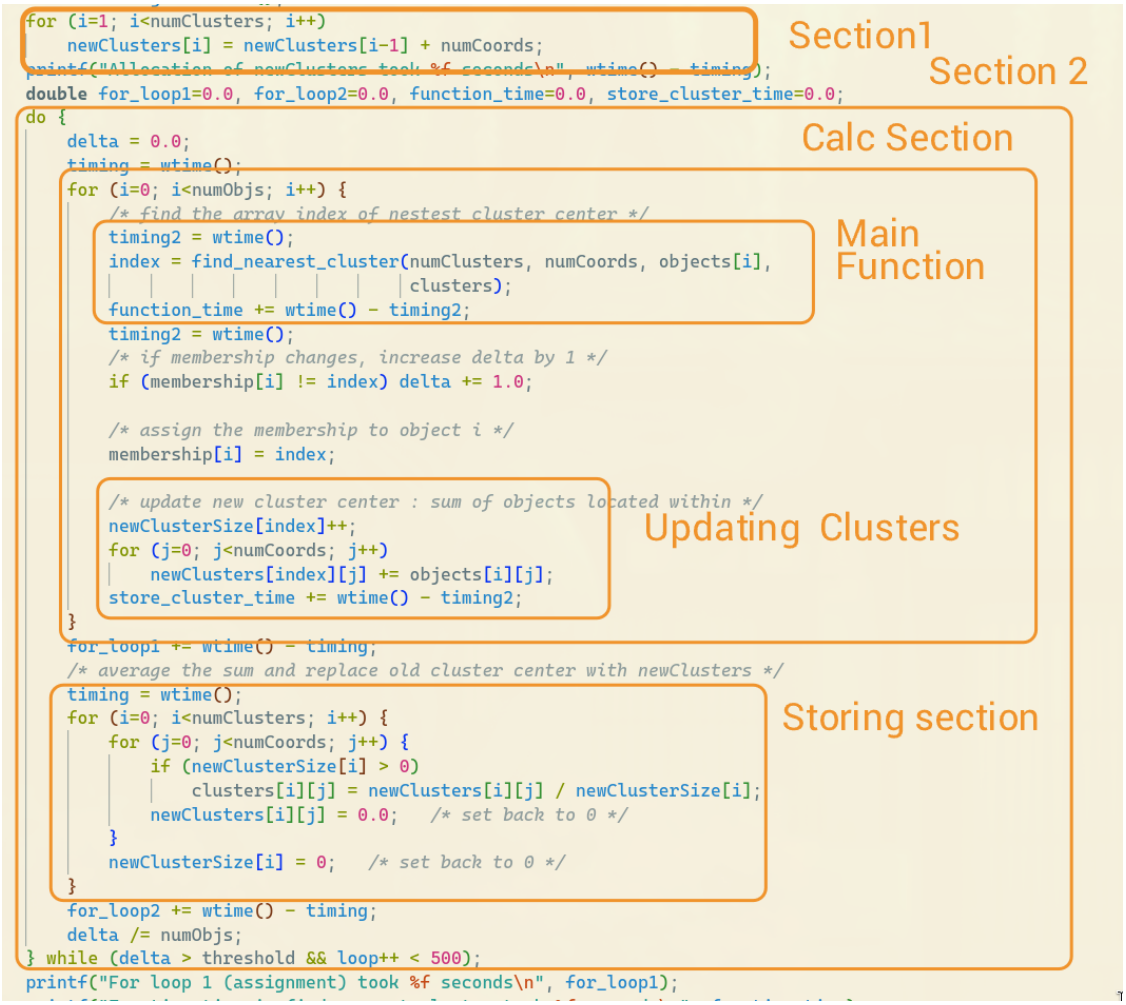

Figure 1: Profiling Sections split up

Table 1: Execution time breakdown of k-means algorithm components

Process	Time (s)
Allocation of `newClusters`	0.000004
For loop 1 (assignment)	3.362360
Function `find_nearest_cluster`	3.339884
Time updating `newClusters`	0.007951
For loop 2 (storing centers)	0.000471

From the above table, we can determine that the main bottleneck of the application is inside the function find_nearest_clusters(). Please note we did not measure standard deviation on these times since it is pretty obvious which part is by far the bigger contributor to the execution time.

4 Optimizations

As discussed earlier the most important part to parallelize is the calculation of the minimum distance between clusters and objects. We notice that each object's calculation for minimum distance can be done in parallel. Hence we will be doing all the distance calculations in parallel.

```
#pragma omp parallel private(i, j, index)
{
    //initializaze local variables
    int *localNewClusterSize = (int *) calloc(numClusters, sizeof(int));
    double **localNewClusters = (double **) malloc(numClusters * sizeof(double *));
    for (i = 0; i < numClusters; i++) {
        localNewClusters[i] = (double *) calloc(numCoords, sizeof(double));
    }
```

Local Memory initialization

```
    #pragma omp for reduction(+:delta) schedule(guided,10)
    for (i=0; i<numObjs; i++) {
        /* find the array index of nestest cluster center */
        index = find_nearest_cluster(numClusters, numCoords, objects[i],
                                     clusters);

        /* if membership changes, increase delta by 1 */
        if (membership[i] != index) delta += 1.0;

        /* assign the membership to object i */
        membership[i] = index;

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

Parallel Calculations

```
    #pragma omp critical
    {
        for (i = 0; i < numClusters; i++) {
            newClusterSize[i] += localNewClusterSize[i];
            for (j = 0; j < numCoords; j++)
                newClusters[i][j] += localNewClusters[i][j];
        }
    }
```

Critical section for calculating new clusters

```
    //free local variables
    for (i = 0; i < numClusters; i++)
    free(localNewClusters[i]);
    free(localNewClusterSize);
    free(localNewClusters);

}
```

Freeing memory allocations

Figure 2: The minimum distance code parallelized.

As you can see we have a parallel region inside the do while loop. This way we parallelize all the nearest cluster calculations in order to gain performance.

4.1 Memory Allocation

One way to gain performance is for each object thread to have its own piece of memory to do calculate the new cluster size for each object. This way we can avoid both false misses and race conditions when adding to a shared variable. It is also worth noting that in OMP, memory allocations inside a parallel region makes the array private.

4.2 Minimum Distance Calculations

To calculate the minimum distance we parallelize the for loop. Each thread works on a single group of objects. It then calculates its new local cluster sum based on the nearest cluster found. We also take great caution when incrementing the shared variable delta, to increment it with the *reduction(+:delta)* directive from Open MP. This way, OMP creates a new local variable for each thread's delta and sums them at the end with a reduction algorithm of its choice to push the results to the global delta variable. Finally we add our own local sum for the newCluster size and the local new clusters.

4.3 New cluster size calculations / Critical region

As discussed in the previous section, the accumulation of the cluster is local to each thread. At the end of each parallel region, the results of each thread's calculations are merged into the shared newCluster variables. This can probably also be done with a reduction algorithm but for now it has been left as is.

4.4 Replacing the old cluster centers with the new ones

```
    /* average the sum and replace old cluster center with newClusters */
    #pragma omp parallel for private(j) schedule(guided,10)
    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 */
    }
    #pragma omp single
    delta /= numObjs;
} while (delta > threshold && loop++ < 500);
```

Figure 3: The code that calculates the new clusters and replaces the old clusters

This part of the code is not very computationally intensive but its still good practice to parallelize it. Here we just let each thread work on averaging the sum of each cluster and then updating it.

4.5 Further optimizations

4.5.1 For loop scheduling

One of the biggest contributors to speedup was the scheduling algorithm that was used. using the guided scheduling which makes the threads to get an exponentially smaller size of work each time decreased the execution time from 0.38s to 0.23 seconds. After some trial and error we came to the conclusion that the best scheduling algorithm was using the guided algorithm with a minimum piece of work of 10.

4.5.2 Better memory allocation

```
int **localClusterSizes = (int **) malloc(nthreads * sizeof(int *));
double ***localClusters = (double ***) malloc(nthreads * sizeof(double **));

for (int t = 0; t < nthreads; t++) {
    localClusterSizes[t] = (int *) malloc(numClusters * sizeof(int));
    localClusters[t] = (double **) malloc(numClusters * sizeof(double *));
    for (i = 0; i < numClusters; i++)
        localClusters[t][i] = (double *) malloc(numCoords * sizeof(double));
}

/* use openMP to parallelize*/
do {
    delta = 0.0;

    #pragma omp parallel private(i, j, index)
    {
        //initialiaze local variables
        int tid = omp_get_thread_num();
        for (i = 0; i < numClusters; i++) {
            localClusterSizes[tid][i] = 0;
            for (j = 0; j < numCoords; j++)
                localClusters[tid][i][j] = 0.0;
        }
```

Figure 4: Code that allocates memory for each thread at the start. Then each thread clears its respective memory.

A part of the code that could be improved was the constant memory allocations for each thread for each iteration of the algorithm. Instead of clearing and allocating every iteration and wasting time, we could allocate only once at the start of the function and then just letting each thread clear it's allocations on its own without the help of calloc. This way we gained a 1.1X improvement on our code, reducing the execution time form 0.26s to 2.23s when using 28 cores.

5 Results

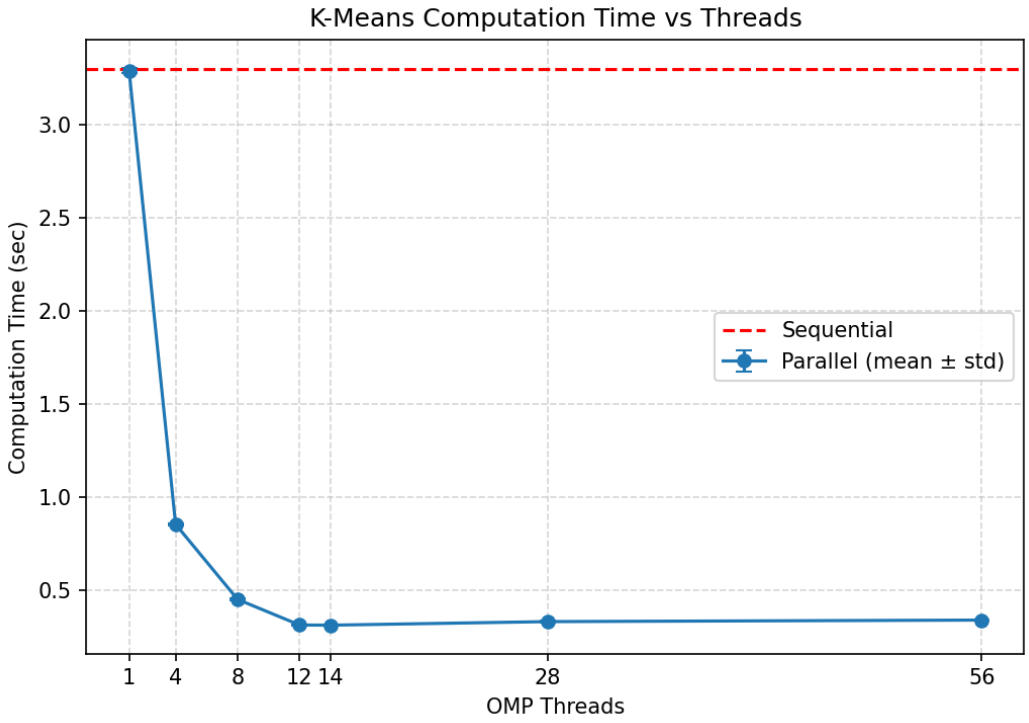

Figure 5: Execution time of of the program on our local computer

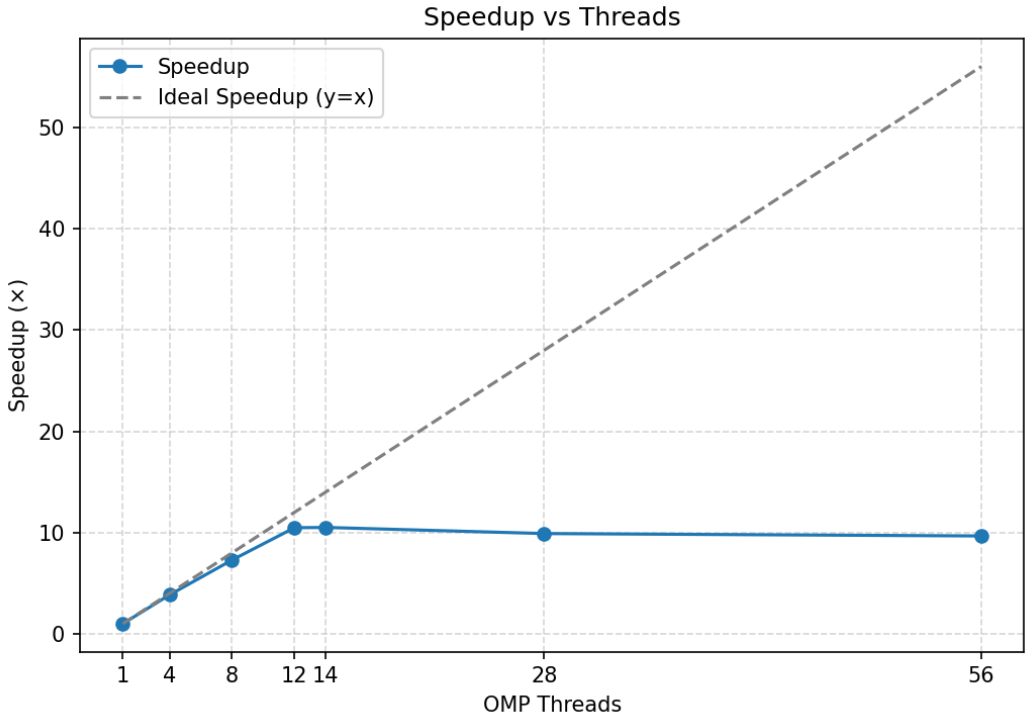

Figure 6: Speedup on our local computer

As you can see the speedup falls very close to the ideal speedup line for programs with 12 or less threads. This is because the CPU that was used contains 12 big cores (since its a binned down version of the 7950x3D with 16 cores). With 12 cores we get a total speedup of 10.5X.

5.1 HPC computer timing

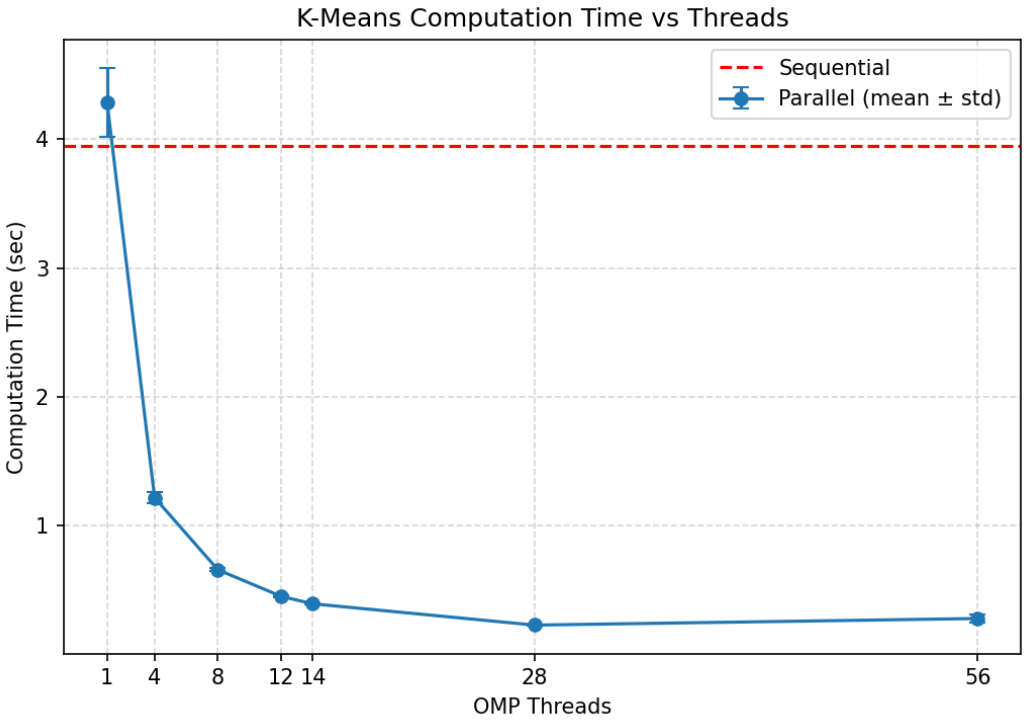

Figure 7: Execution time of of the program on the HPC computer

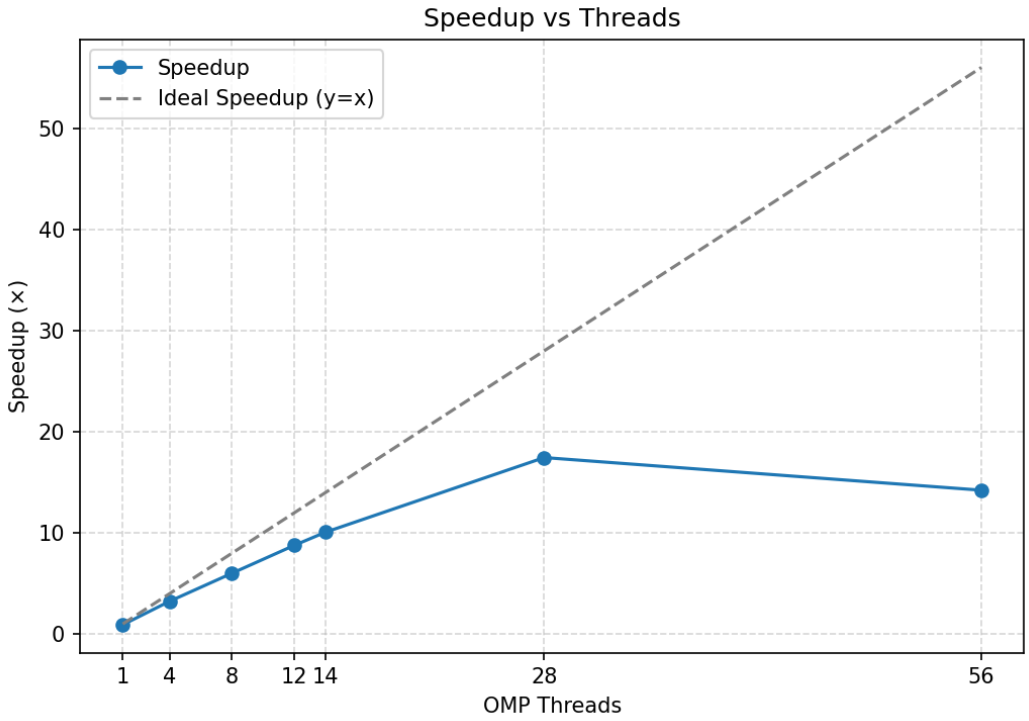

Figure 8: Speedup on the HPC computer

Table 2: K-Means Execution Time vs Thread Count

Program	**Threads**	**Mean Time (s)**	**Std. Dev. (s)**
SEQ	0	3.9448	0.1084
PAR	1	4.2858	0.2702
PAR	4	1.2162	0.0443
PAR	8	0.6557	0.0118
PAR	12	0.4493	0.0091
PAR	14	0.3914	0.0061
PAR	28	0.2261	0.0041
PAR	56	0.2774	0.0323

As for the HPC machine, the speedup was a marginally worse per core we used. Instead of getting close to the Px speedup where P is the number of processors, at 28 cores used, we gained a total of 17.3X speedup. This can be attributed to the Xeon being older and having smaller cores in comparison with the Ryzen CPU.

6 Conclusion

On our local Ryzen 9 7900X3D system, the implementation achieved a maximum speedup of approximately 10.5× with 12 threads, nearly linear scaling up to the number of physical cores. On the HPC Xeon-based system, scaling was still significant, reaching a 17.3× speedup at 28 cores, though with slightly lower efficiency due to architectural differences and smaller per-core performance.

Overall, in this lab, we saw the speedup that can be gained by parallelizing a code by utilizing Open MP.