ECE415: High Performance Computing Systems
Lab2 - Clustering k-means Parallelization with OpenMP

Contents

1 Introduction

The focus of this lab is to optimize the K-means clustering algorithm. The algorithm works as follows: it assigns each data point to its nearest centroid and then recalculates the centroid based on the newly assigned points. In each iteration, the code computes the Euclidean distances between all objects and all centroids, updates each object's membership and accumulates the partial sums and counts for each cluster. After that it updates each cluster's centroid by dividing the accumulated sums by the counts and resets them to zero for the next iteration. This repeats until either the defined maximum number of iterations is reached or the fraction of objects whose membership changed falls below a defined threshold.

2 Compiler

Compilation was done using icx, Intel C compiler.
The following optimization flags were tested on the serial code:

```
-O0
-O2
-O3
-O3 -ffast-math
-O3 -xHost
-O3 -ipo -xHost
-O3 -ffast-math -ipo -xHost
```

After running the program 12 times for each of the flags above, calculating their average execution times, having removed their lowest and highest values, the flags chosen for their best execution time and thus used for all our subsequent experiments are:

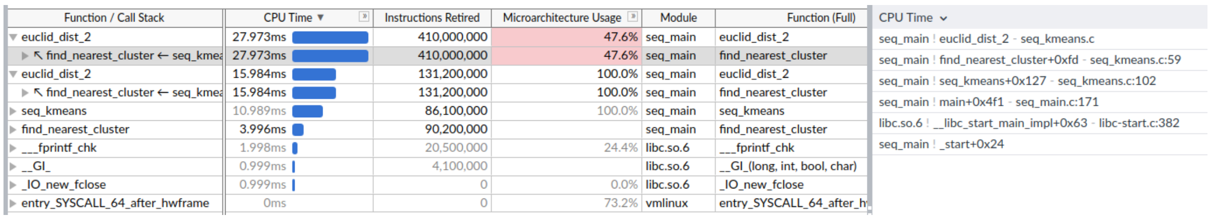
```
-O3 -ffast-math
```

The program was tested by running:

```
./seq\_main -o -b -n 2000 -i Image\_data/texture17695.bin
```

3 Code Profiling

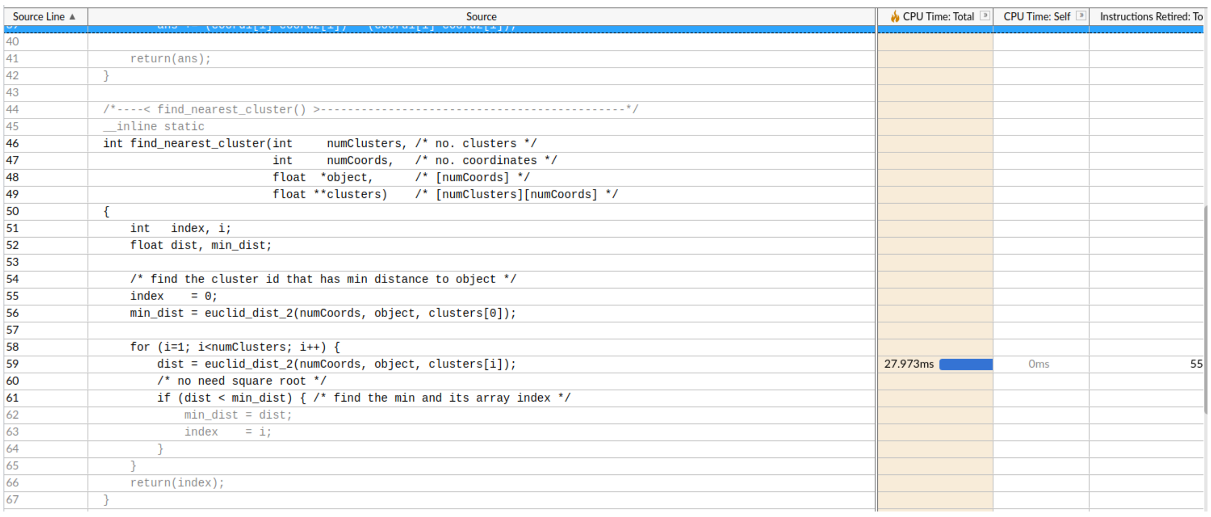
Intel vtune was used to profile the code and find the functions and operations were the serial implementation spent the most time executing.



Function / Call Stack	CPU Time ▼	Instructions Retired	Microarchitecture Usage	Module	Function (Full)
▼ euclid_dist_2	27.973ms	410,000,000	47.6%	seq_main	euclid_dist_2
▶ ↖ find_nearest_cluster ← seq_kmea	27.973ms	410,000,000	47.6%	seq_main	find_nearest_cluster
▼ euclid_dist_2	15.984ms	131,200,000	100.0%	seq_main	euclid_dist_2
▶ ↖ find_nearest_cluster ← seq_kmea	15.984ms	131,200,000	100.0%	seq_main	find_nearest_cluster
▶ seq_kmeans	10.989ms	86,100,000	100.0%	seq_main	seq_kmeans
▶ find_nearest_cluster	3.996ms	90,200,000		seq_main	find_nearest_cluster
▶ ___fprintf_chk	1.998ms	20,500,000	24.4%	libc.so.6	___fprintf_chk
▶ __GI_	0.999ms	4,100,000		libc.so.6	__GI_(long, int, bool, char)
▶ _IO_new_fclose	0.999ms	0	0.0%	libc.so.6	_IO_new_fclose
▶ entry_SYSCALL_64_after_hwframe	0ms	0	73.2%	vmlinux	entry_SYSCALL_64_after_h

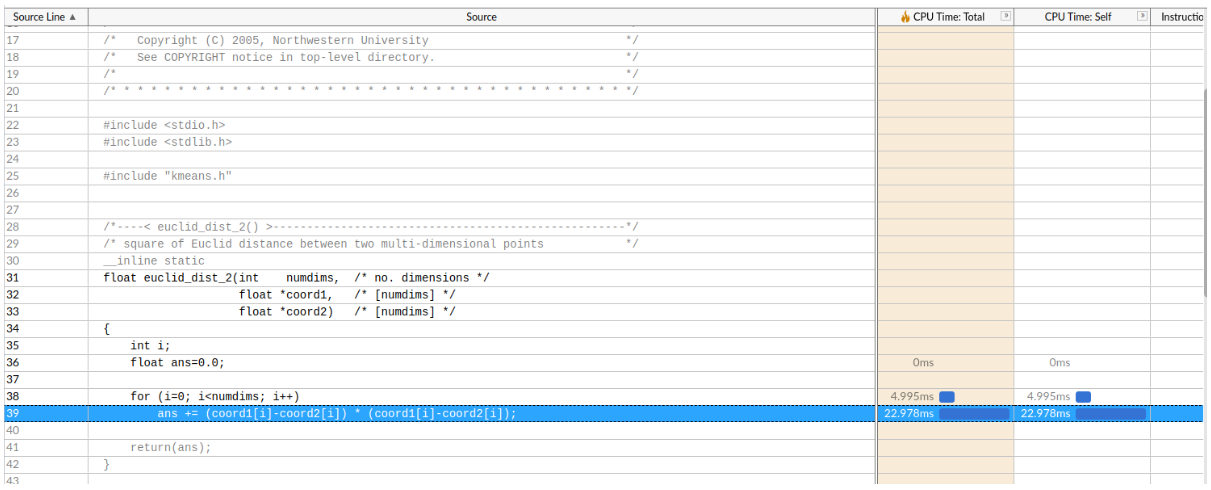
CPU Time ⌄
seq_main ! euclid_dist_2 - seq_kmeans.c
seq_main ! find_nearest_cluster+0xfd - seq_kmeans.c:59
seq_main ! seq_kmeans+0x127 - seq_kmeans.c:102
seq_main ! main+0x4f1 - seq_main.c:171
libc.so.6 ! __libc_start_main_impl+0x63 - libc-start.c:382
seq_main ! _start+0x24

Figure 1: Execution Time per Function.



```
Source Line  Source                                                                              CPU Time: Total  CPU Time: Self  Instructions Retired: To
40
41               return(ans);
42           }
43
44           /*----< find_nearest_cluster() >-------------------------------------------*/
45           __inline static
46           int find_nearest_cluster(int     numClusters, /* no. clusters */
47                                    int     numCoords,   /* no. coordinates */
48                                    float  *object,      /* [numCoords] */
49                                    float **clusters)    /* [numClusters][numCoords] */
50           {
51               int   index, i;
52               float dist, min_dist;
53
54               /* find the cluster id that has min distance to object */
55               index    = 0;
56               min_dist = euclid_dist_2(numCoords, object, clusters[0]);
57
58               for (i=1; i<numClusters; i++) {
59                   dist = euclid_dist_2(numCoords, object, clusters[i]);                 27.973ms         0ms             55
60                   /* no need square root */
61                   if (dist < min_dist) { /* find the min and its array index */
62                       min_dist = dist;
63                       index    = i;
64                   }
65               }
66               return(index);
67           }
```

Figure 2: find_nearest_cluster hot-spot.



```
Source Line  Source                                                                              CPU Time: Total  CPU Time: Self  Instructio
17           /*   Copyright (C) 2005, Northwestern University                          */
18           /*   See COPYRIGHT notice in top-level directory.                         */
19           /*                                                                        */
20           /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
21
22           #include <stdio.h>
23           #include <stdlib.h>
24
25           #include "kmeans.h"
26
27
28           /*----< euclid_dist_2() >--------------------------------------------------*/
29           /* square of Euclid distance between two multi-dimensional points         */
30           __inline static
31           float euclid_dist_2(int    numdims,  /* no. dimensions */
32                               float *coord1,   /* [numdims] */
33                               float *coord2)   /* [numdims] */
34           {
35               int i;
36               float ans=0.0;                                                                  0ms              0ms
37
38               for (i=0; i<numdims; i++)                                                       4.995ms          4.995ms
39                   ans += (coord1[i]-coord2[i]) * (coord1[i]-coord2[i]);                       22.978ms         22.978ms
40
41               return(ans);
42           }
43
```

Figure 3: euclid_dist_2 hot-spot.

As shown in the images above, we observe that most of the time spent is in the function `euclid_dist_2`.

4 Parallelization Attempts

4.1 Optimization Attempts for euclid_dist_2

As the function `euclid_dist_2` seemed to dominate the execution time, we started our optimizations efforts there. The most time is apparently spent in the loop of the `euclid_dist_2` function below:

```
    for (i=0; i<numdims; i++)
        ans += (coord1[i]-coord2[i]) * (coord1[i]-coord2[i]);
```

4.1.1 1st Optimization Attempt -Dismissed

That is why we tried to parallelize this loop as following using `reduction(+:ans)` so that there are no race conditions for the variable `ans`.

```
    #pragma omp parallel for reduction(+:ans)
    for (i=0; i<numdims; i++)
        ans += (coord1[i]-coord2[i]) * (coord1[i]-coord2[i]);
```

That not only did not work but actually worsened the execution time considerably. This is likely because `numdims` is of small size(20), so parallelizing this loop is probably creating a huge overhead by creating and destroying threads very often.

4.1.2 2nd Optimization Attempt -Dismissed

Another approach was to use `simd reduction` instead of parallel as following:

```c
    #pragma omp simd reduction(+:ans)
    for (i=0; i<numdims; i++)
        ans += (coord1[i]-coord2[i]) * (coord1[i]-coord2[i]);
```

That did not have any effect on the execution time, so it was also dismissed.

4.2 Optimization Attempts for find_nearest_cluster

Based on VTune analysis we identified `find_nearest_cluster` as the next hotspot and chose to optimize it next. This function scans all centroids, computes the squared Euclidean distance to each by calling `euclid_dist_2` and returns the index of the minimum distance.

4.2.1 1st Optimization Attempt

Since each distance computation is independent, we can parallelize the loop over centroids. To avoid race conditions, each thread has its own local distance, local best distance and local index. Each thread calculates its locals in parallel which are then merged in a critical section to produce the global minimum. To be more exact we create an omp parallel region that includes:

- `#pragma omp for nowait`: Each thread computes a subset of centroids using its own local variables (as the computation is independent across iterations). Here we use the `nowait` clause so that the the implicit barrier at the end of this `for` is removed and any thread finished can move on to the `#pragma omp critical` section without having to wait for all the threads to finish.
- `#pragma omp critical`: It is used to safely merge the local variables of each thread into the shared global minimum (distance and index).

```c
int find_nearest_cluster(int numClusters, int numCoords, float *object, float **clusters
    ){
    int index = 0;
    float dist = euclid_dist_2(numCoords, object, clusters[0]);

    #pragma omp parallel
    {
        int local_index = 0;
        float local_best_dist = dist;

        #pragma omp for nowait
        for (int i = 1; i < numClusters; i++) {
            float dist = euclid_dist_2(numCoords, object, clusters[i]);
            if (dist < local_best_dist) {
                local_best_dist = dist;
                local_index = i;
            }
        }

        #pragma omp critical
        {
            if (local_best_dist < dist) {
                dist = local_best_dist;
                index = local_index;
            }
        }
    }
    return index;
}
```

The following graphs (Figure 4, Figure 5) show the average execution times for 12 runs of this optimization (excluding the maximum and minimum values) for thread counts of 1, 4, 8, 14, 28, and 56, along with their standard deviations.

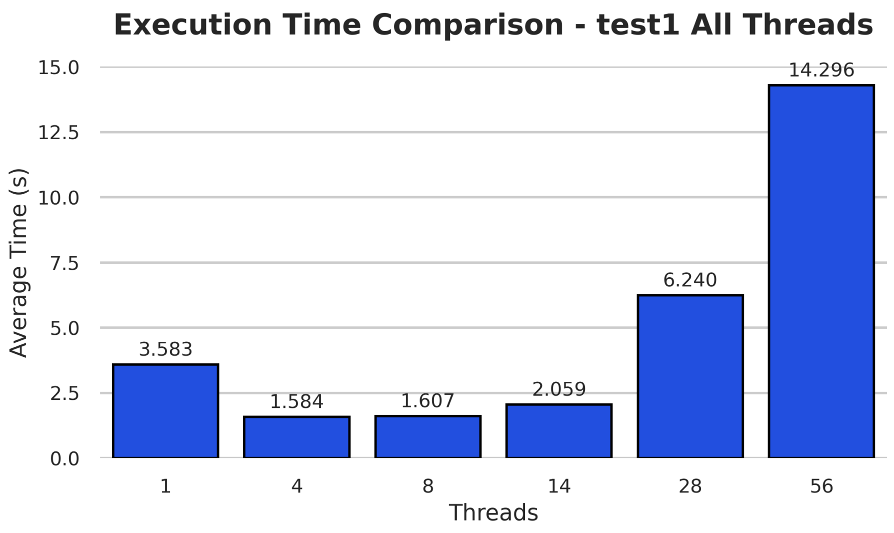


Figure 4: Execution times for various thread counts (find_nearest_cluster - 1st attempt)

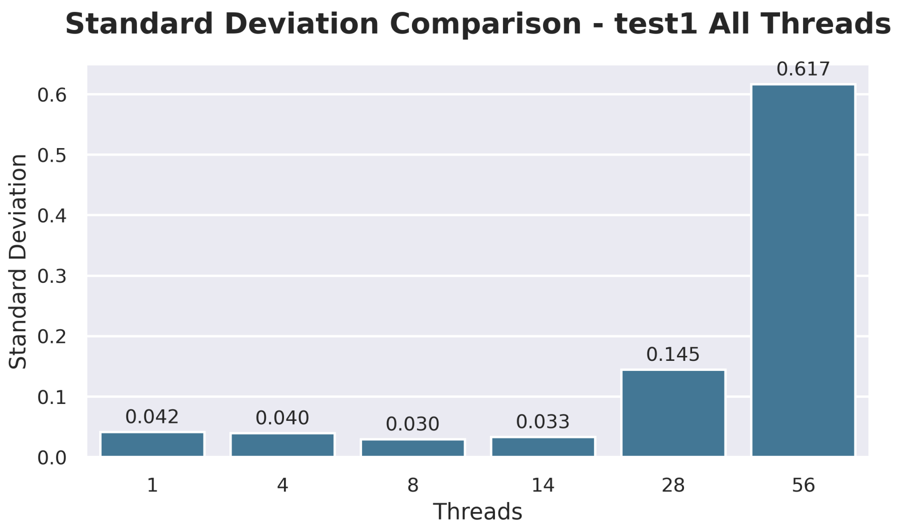


Figure 5: Standard deviations of execution times for thread counts (find_nearest_cluster - 1st attempt)

4.2.2 2nd Optimization Attempt -Dismissed

This next optimization attempt on `find_nearest_cluster()` aimed to eliminate the need for `#pragma omp critical` section by and restructuring the computation as follows:

1. **Array allocation:** Allocate a temporary `float` array (`dist_array`) of size `numClusters` to store the Euclidean distance between the current object and each cluster center.
2. **Parallel computation:** Use `#pragma omp parallel for` so that each thread computes `euclid_dist_2()` for a different cluster index `i`. Each thread writes to a unique array location (`dist_array[i]`) which prevents race conditions and avoids the synchronization inside the parallel region.
3. **Sequential reduction:** After all distances are computed, perform a simple sequential scan outside the parallel region to find the smallest distance and corresponding cluster index. Since the array size is relatively small (`numClusters = 2000`) this scan should in theory be quick enough to offset the synchronization cost of the previous optimization.

However the fact that the parallel loop's performance is limited by the slowest thread, makes this implementation slower accross all threas compared to the previous one using `#pragma omp critical` so it was dismissed.

```c
int find_nearest_cluster(int numClusters, int numCoords, float *object, float **clusters
    )
{
    int i;
    float *dist_array;
    int min_index = 0;
    float min_dist;
    dist_array = (float*) malloc(numClusters * sizeof(float));
    dist_array[0] = euclid_dist_2(numCoords, object, clusters[0]);

    #pragma omp parallel for
    for (i = 1; i < numClusters; i++) {
        dist_array[i] = euclid_dist_2(numCoords, object, clusters[i]);
    }

    min_dist = dist_array[0];
    min_index = 0;

    for (i = 1; i < numClusters; i++) {
        if (dist_array[i] < min_dist) {
            min_dist = dist_array[i];
            min_index = i;
        }
    }

    free(dist_array);
    return min_index;
}
```

4.3 Optimization Attempts for seq_kmeans

The original `seq_kmeans` function processes all objects sequentially, finds the nearest cluster for each object, updates the membership and accumulates cluster sums and counts. The cluster centers are then updated by averaging the accumulated sums.

4.3.1 1st Optimization Attempt -Dismissed

To improve performance, the loop over objects was parallelized using OpenMP with the following changes in the parallel region:

1. The variable `delta` which tracks the number of membership changes and is shared across the threads is now in a `reduction(+:delta)` clause. Now every thread has a private copy of `delta` during execution that is safely combined at the end of the parallel region.

2. The variables `index` and `j` defined outside the parallel region are now declared as `private` so that threads avoid data races.

```c
#pragma omp parallel for reduction(+:delta) private(index, j)
for (int i = 0; i < numObjs; i++) {
    index = find_nearest_cluster(numClusters, numCoords, objects[i],
                                                 clusters);

    if (membership[i] != index) delta += 1.0;

    membership[i] = index;
    // ...
}
```

3. Updates to the shared cluster accumulators `newClusterSize` and `newClusters` are protected by an OpenMP `critical` section to avoid data races during concurrent updates.

```c
#pragma omp critical
{
    newClusterSize[index]++;
    for (j = 0; j < numCoords; j++)
```

```
        newClusters[index][j] += objects[i][j];
}
```

4. After this parallel region, we have another separate parallel loop that averages the accumulated sums to update the cluster centers and resets the accumulators for the next iteration. This loop used to compute the new cluster centroids is parallelized with `#pragma omp parallel for` since the computations for each cluster are independent.

```
#pragma omp parallel for
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;   /* reset for next iteration */
    }
    newClusterSize[i] = 0;   /* reset for next iteration */
}
```

Output Discrepancies

At all stages of our optimizations we verified the correctness of our code by comparing the output files produced by our parallel optimized versions (*.bin.cluster_centers, *.bin.membership) with their respective golden versions produced by running the original sequential code. At this point a very minimum error was observed, in the comparison of the the *.bin.cluster_centers files while the *.bin.membership files remained identical. This minor deviation was introduced due to floating arithmetic operations performed during the computation of new cluster centers, specifically the repeated addition and division of floating point numbers when accumulating object coordinates and averaging them within each cluster which are made in a non deterministic order in our parallel versions. Each line in the *.bin.cluster_centers file represents a single cluster center. The first value corresponds to the cluster index followed by its feature coordinates (floating point values). When comparing the sequential and parallel outputs, the observed discrepancies between corresponding coordinate values were on the order of 10^{-6}.This difference represents the absolute error, defined as $|x_{\text{parallel}} - x_{\text{sequential}}|$ and is considered negligible.

The following graphs (Figure 6, Figure 7) show the average execution times for 12 runs of this optimization (excluding the maximum and minimum values) for thread counts of 1, 4, 8, 14, 28, and 56, along with their standard deviations.

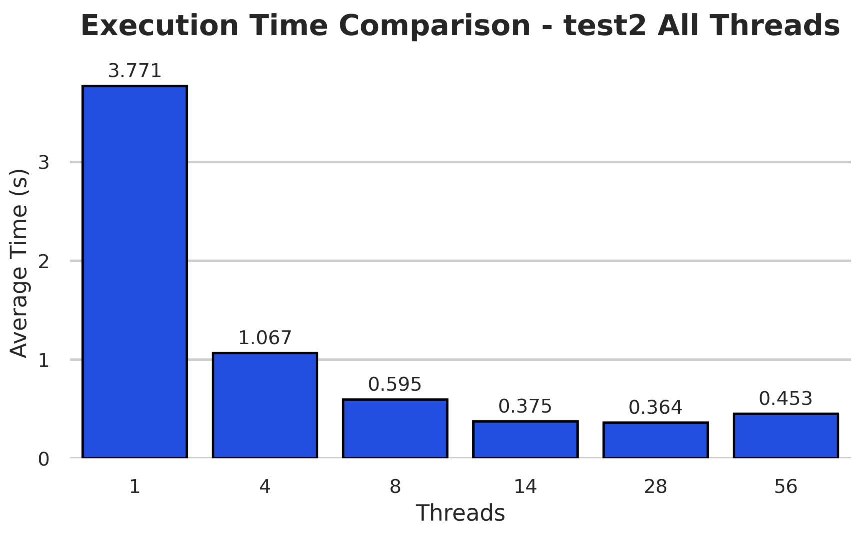


Figure 6: Execution times for various thread counts (seq_kmeans - 1st attempt)

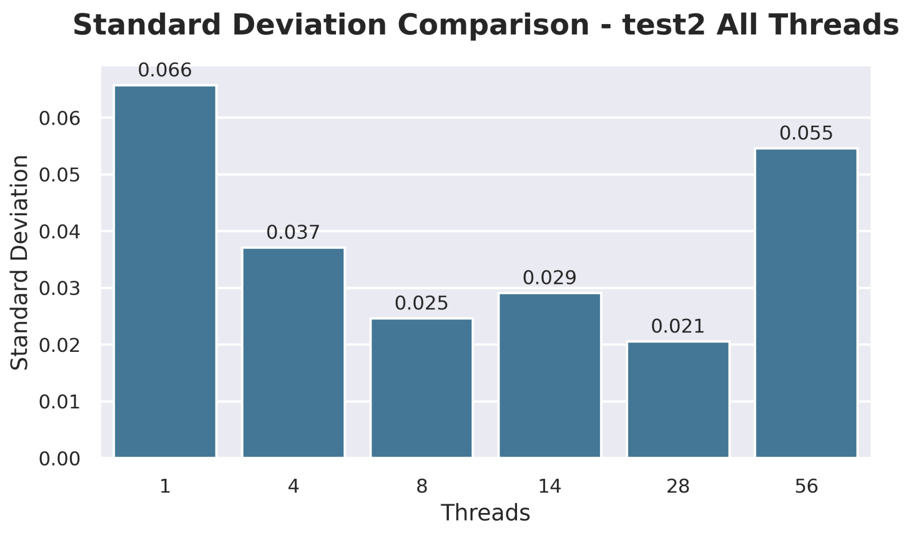


Figure 7: Standard deviations of execution times for thread counts (seq_kmeans - 1st attempt)

4.3.2 2nd Optimization Attempt

In the final set of optimizations, we tried to limit the critical section so that more work can be executed in parallel. To achieve this we used a similar approach to section 4.2.1 declaring local variables in the parallel region, so that each thread can perform its own local accumulation and after in a now smaller critical section calculate the globals, summing all the thread's accumulated locals. Here we also experimented with scheduling policies for our *parallel for loops*. To be more specific we did the following:

1. **Thread-local accumulation:** Within the parallel region, each thread uses *thread-local* buffers for partial cluster sums and accounts (`**localClusters`, `*localClusterSize` per thread) and its local delta too (`localDelta`). This avoids race conditions during the local accumulation, while allowing more work to be parallelized.

```c
int *localClusterSize = (int*) calloc(numClusters, sizeof(int));
float **localClusters  = (float**) malloc(numClusters * sizeof(float*));
localClusters[0] = (float*) calloc(numClusters * numCoords, sizeof(float));

for (i = 1; i < numClusters; i++)
    localClusters[i] = localClusters[0] + i * numCoords;

    float localDelta = 0.0;

```

2. **Safe global merge:** After processing its chunk, each thread merges its partial results into the global accumulators. We used `#pragma omp atomic` to safely accumulate `delta` and a short `#pragma omp critical` block to merge partial cluster sums and sizes into `**newClusters` and `*newClusterSize`.

```c
#pragma omp atomic
delta += localDelta;

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

```

3. **Scheduling:** After experimenting with different scheduling policies for our *parallel for loops*, we discovered that `schedule(dynamic, 25)` had the best time results for all said loops, potentially because it can handle the load imbalance across objects. To reach this conclusion we tested each

parallel for loop with static scheduling, as well as dynamic and guided with various chunk sizes (100, 75, 50, 25, 20).

4. **Remove inner parallelism:** We removed the parallelization inside `find_nearest_cluster` and returned it to its sequential form, after observing that it provided no benefit and in fact introduced additional overhead at this point. The outer level parallelism over objects seemed to provide better granularity and scaling.

```c
    int find_nearest_cluster(int numClusters, int numCoords,
                             float *object, float **clusters)
{
    int   index = 0;
    float min_dist = euclid_dist_2(numCoords, object, clusters[0]);

    for (int i = 1; i < numClusters; i++) {
        float dist = euclid_dist_2(numCoords, object, clusters[i]);
        if (dist < min_dist) {
            min_dist = dist;
            index    = i;
        }
    }
    return index;
}

```

5. **Use of privates:** As in previous optimizations we used `private(...)`, where needed to protect variables defined outside the parallel region that might have race conditions.

6. **Initialization of newClusters:** In order to parallelize the initialization of newClusters, because of the way it was written each element was dependent on the previous one (newClusters[i] = newClusters[i-1] + numCoords;), we changed the initialization to be done with pointer arithmetic.

```c
    #pragma omp parallel for schedule(dynamic, 25)
    for (i=1; i<numClusters; i++)
        newClusters[i] = newClusters[0] + i*numCoords;

```

Output Discrepancies

The output discrepancies between the files *.bin.cluster_centers mentioned in section 4.3.1 were also observed here due to the same reasons.

Following is the code for seq_kmeans, so that overall details of its structure can be observed:

```c
do {
        delta = 0.0;

        /* parallel region: each thread works on a subset of objects */
        #pragma omp parallel private(i, j, index)
        {
            int *localClusterSize = (int*) calloc(numClusters, sizeof(int));
            float **localClusters = (float**) malloc(numClusters * sizeof(float*));
            localClusters[0] = (float*) calloc(numClusters * numCoords, sizeof(float));
            for (i=1; i<numClusters; i++)
                localClusters[i] = localClusters[0] + i*numCoords;

            float localDelta = 0.0;

            #pragma omp for schedule(dynamic, 25) nowait
            for (i=0; i<numObjs; i++) {
                index = find_nearest_cluster(numClusters, numCoords, objects[i],
                                             clusters);


                if (membership[i] != index) localDelta += 1.0;

                membership[i] = index;

                /* update local cluster center : sum of objects located within */
                localClusterSize[index]++;
                for (j=0; j<numCoords; j++)
                    localClusters[index][j] += objects[i][j];
            }

            /* reduction step merge thread local partial results into global */
            #pragma omp atomic
            delta += localDelta;

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

            free(localClusterSize);
            free(localClusters[0]);
            free(localClusters);
        } /* end parallel region */

        /* average the sum and replace old cluster center with newClusters */
        #pragma omp parallel for private(j) schedule(dynamic, 25)
        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 */
        }

        delta /= numObjs;
    } while (delta > threshold && loop++ < 500);
```

Listing 1: K-means thread-local accumulation

The following graphs (Figure 8, Figure 9) show the average execution times for 12 runs of this optimization (excluding the maximum and minimum values) for thread counts of 1, 4, 8, 14, 28, and 56, along with their standard deviations.

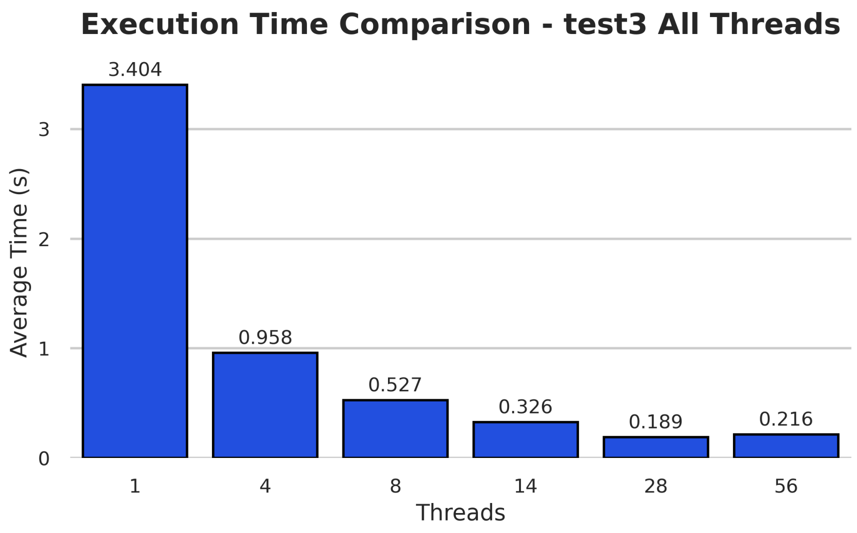


Figure 8: Execution times for various thread counts (seq_kmeans - 2nd attempt)

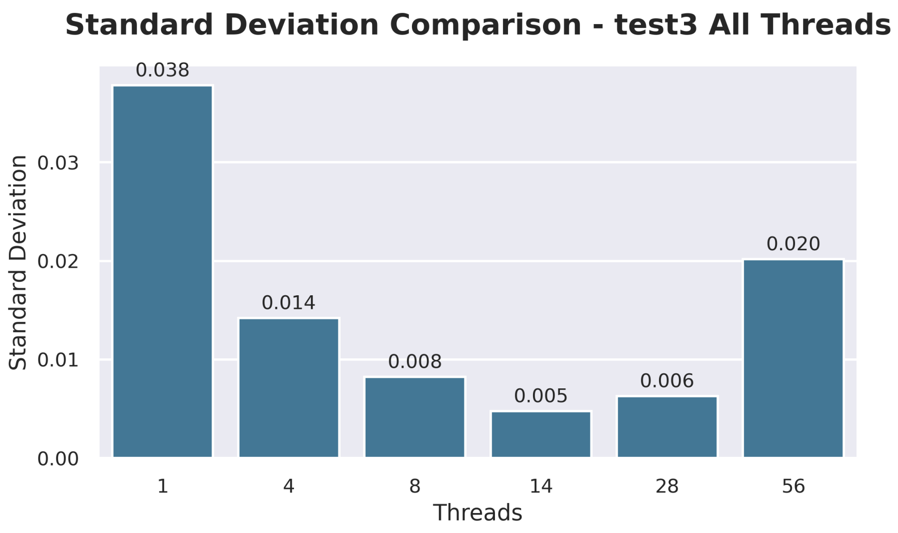


Figure 9: Standard deviations of execution times for thread counts (seq_kmeans - 2nd attempt)

4.3.3 3rd Optimization Attempt - Dismissed

A final optimization we tried was to extend the idea used in the previous optimization and implement a thread-local storage for all cluster accumulations with a separate buffer allocated per thread. This allows each thread to work independently doing its own local computation and merging the results with the other threads at the very end.

To be more specific, each thread gets its own copy of the data structures that store partial cluster information, so we added one more level of indexing to our arrays. In the sequential version we had these two arrays `**newClusters` and `*newClusterSize` and there was only one copy of each. Now to run fully in parallel we give each thread its own version of those arrays by creating arrays of them like this: `**localClusterSizes` and `***localClusters` and a local_delta as well.

We do this by first calculating how many threads the program uses, to find out the size of the extra dimension and allocate the space needed. So the extra dimension corresponds to the thread ID and we have one full "mini copy" of the cluster arrays per thread. That way each thread accesses its own section based on its thread ID (`tid = omp_get_thread_num()`) accumulating local cluster sums and counts without race conditions.

At the end of each iteration we combine all these thread-local arrays into the global ones (`newClusters` and `newClusterSize`) in a short critical section. This way most of the heavy computation happens in parallel without synchronization overhead and only a small merging step is done sequentially.

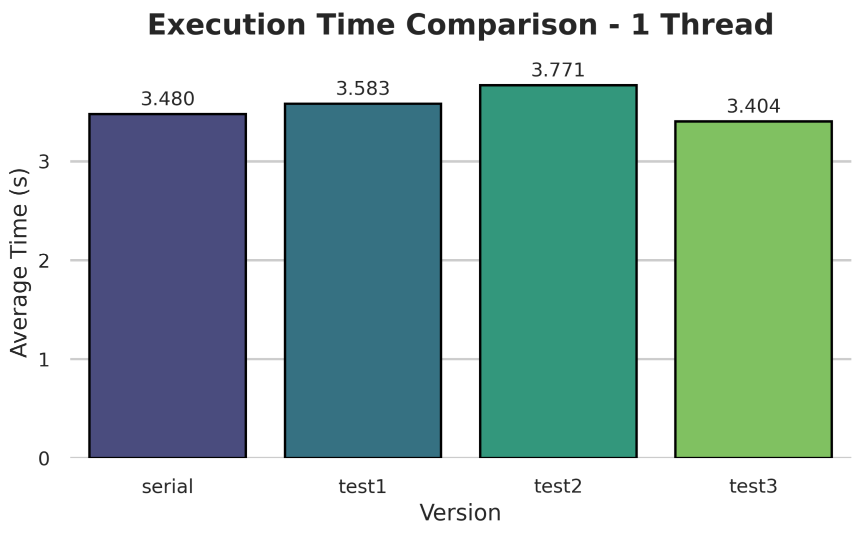
The general parallelization of the for loops follows the same logic as in section : 4.3.2.

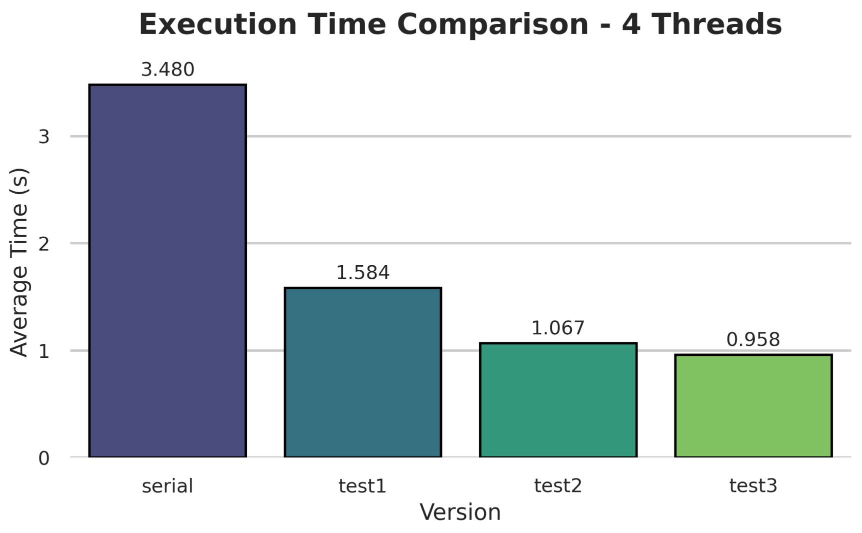
```c
nthreads = omp_get_num_threads();

int **localClusterSizes = (int**) malloc(nthreads * sizeof(int*));
float ***localClusters = (float***) malloc(nthreads * sizeof(float**));

for (int t = 0; t < nthreads; t++) {
    localClusterSizes[t] = (int*) calloc(numClusters, sizeof(int));
    localClusters[t] = (float**) malloc(numClusters * sizeof(float*));
    localClusters[t][0] = (float*) calloc(numClusters * numCoords, sizeof(float));
    for (int k = 1; k < numClusters; k++)
        localClusters[t][k] = localClusters[t][0] + k * numCoords;

do {
        delta = 0.0;

        #pragma omp parallel private(i, j, index)
        {
            int tid = omp_get_thread_num();
            int *myClusterSize = localClusterSizes[tid];
            float **myClusters = localClusters[tid];
            float localDelta = 0.0;

            /* clear thread-local arrays before each iteration */
            for (i = 0; i < numClusters; i++) {
                myClusterSize[i] = 0;
                for (j = 0; j < numCoords; j++)
                    myClusters[i][j] = 0.0;
            }

            #pragma omp for schedule(dynamic,25) nowait
            for (i = 0; i < numObjs; i++) {
                /* find the array index of nearest cluster center */
                index = find_nearest_cluster(numClusters, numCoords, objects[i],
                                    clusters);

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

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

                /* update local cluster center : sum of objects located within */
                myClusterSize[index]++;
                for (j = 0; j < numCoords; j++)
                    myClusters[index][j] += objects[i][j];
            }

         #pragma omp critical
            {
                delta += localDelta;
                for (i = 0; i < numClusters; i++) {
                    newClusterSize[i] += myClusterSize[i];
                    for (j = 0; j < numCoords; j++)
                        newClusters[i][j] += myClusters[i][j];
                }
            }

        ....
        ....
        ....
}
```

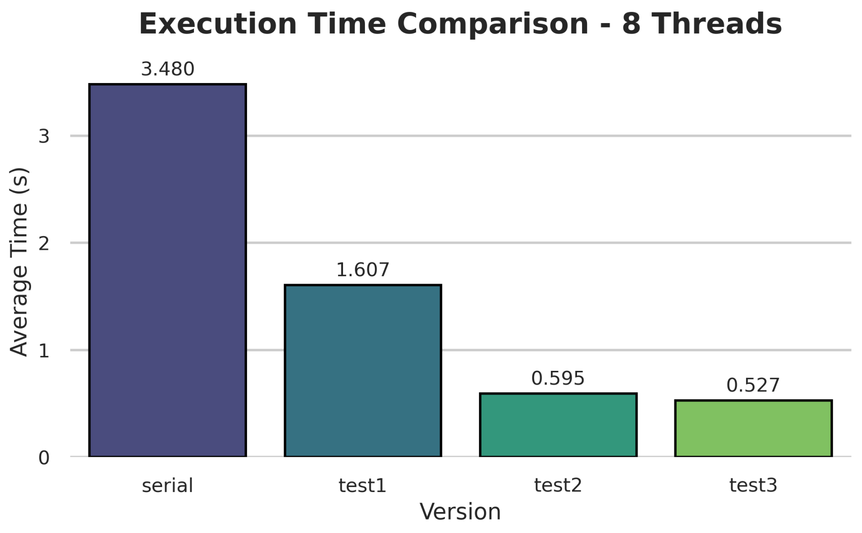
This attempt performed slightly worse in terms of execution time compared to the previous one so it was ultimately dismissed.

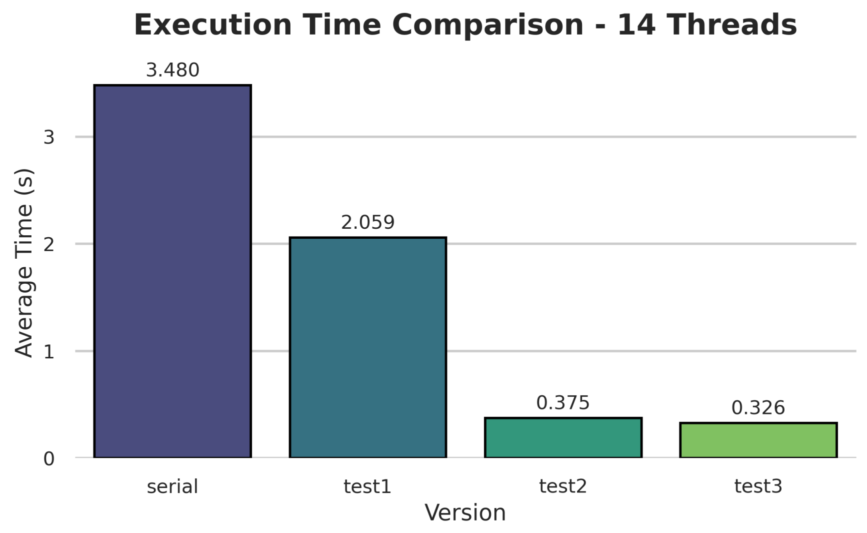
5 Final Experimental Results

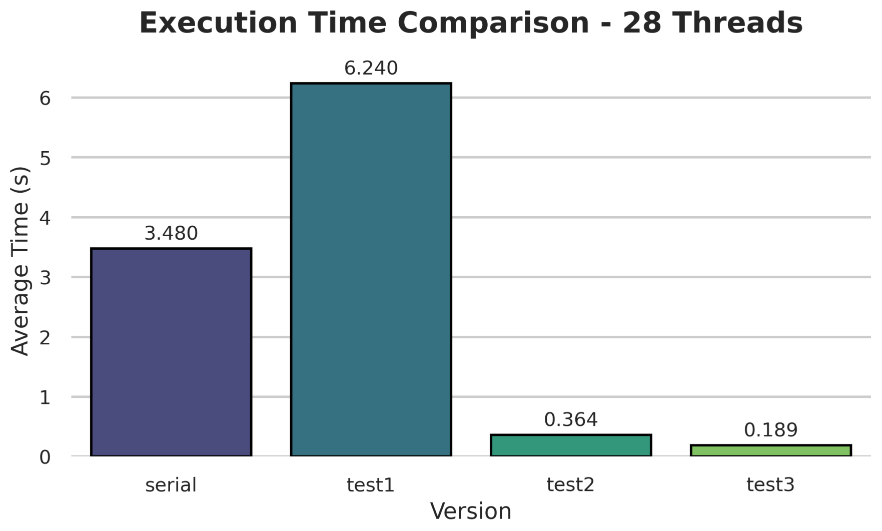
Below are graphs showing the execution times of the useful tests conducted in comparison to one another and the serial version using different numbers of threads.

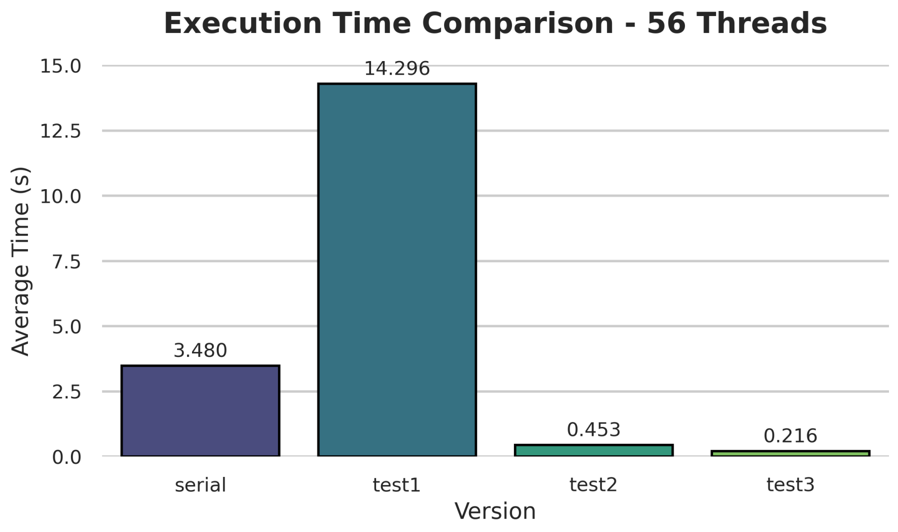

Execution Time Comparison - 1 Thread
3.480
3.583
3.771
3.404
Average Time (s)
0
1
2
3
serial
test1
test2
test3
Version


Execution Time Comparison - 4 Threads
3.480
1.584
1.067
0.958
Average Time (s)
0
1
2
3
serial
test1
test2
test3
Version


Execution Time Comparison - 8 Threads
3.480
1.607
0.595
0.527
Average Time (s)
0
1
2
3
serial
test1
test2
test3
Version







As shown on the graphs above the best execution time is 0.189s (189ms) for test3 (section: 4.3.2) and 28 threads.

6 Conclusion

From our experiments and optimization attempts we observe that parallelizing the large and computationally expensive portions of the code yields significant performance speedup. We also found out that increasing the number of threads improves execution time up to a certain point (from 1 to 28 threads). However using too many threads (56) actually worsens performance due to the increased synchronization overhead and thread management costs. The most important optimizations are:

- **Parallelize large and expensive code sections:**Parallelizing large, compute heavy and loop guided code sections makes the overhead from thread creation, synchronization and management worthwhile compared to the speedup achieved.
- **Avoid parallelizing small functions or small loops:** Parallelizing small or fast executing functions (`euclid_dist_2` or inner loops) results in lower performance due to the overhead crated by the threads.
- **Use thread local variables and accumulators:** Using local variables and buffers for each thread for partial sums and counts instead of shared values (for example with reduce) minimizes contention and synchronization and yields faster execution times.
- **Minimize synchronization overhead:** In the same context as the previous one using minimal critical sections and atomic operations only when necessary to protect shared data.