ECE415 - High Performance Computing

– Homework 2 –

Parallel K-Means Clustering using OpenMP

submitted by

Aggeliki-Ostralis Vliora, Spyridon Liaskonis

November 5, 2024

Aggeliki-Ostralis Vliora, Spyridon Liaskonis
`avliora@uth.gr, sliaskonis@uth.gr`
Student ID: 03140,03381

Contents

1 Introduction

K-means clustering is a widely used algorithm for organizing unlabeled data points into distinct clusters based on similarity. In this project, we aim to boost the efficiency of K-means by parallelizing its computations using OpenMP.

Our experimentation is structured as follows. We begin with a sequential implementation of K-means clustering, profiling the code with tools like Intel's V-Tune Profiler to identify performance bottlenecks. Based on these insights, we incrementally parallelize different parts of the algorithm, testing each optimization step-by-step. After each optimization, we compare performance metrics with previous versions to determine the most effective implementation. If a modification results in decreased performance, we discard it and revert to the previous version.

For testing, we run a standardized test across all optimized versions, executing each on varying thread counts to assess scalability. The number of threads used depends on the target hardware, and in our case, we evaluate each version using 1, 4, 8, 14, 28, and 56 threads.

1.1 Evaluation

1.1.1 Hardware

The code evaluation was performed on a multicore system powered by Intel Xeon E5-2695 processor. This machine features two processors, each with 14 cores, resulting in a 28-core configuration. With support for simultaneous multithreading (SMT), or hyperthreading, each core can manage two threads concurrently, allowing us to parallelize the algorithm across 56 threads.

1.1.2 Software

For software, we used Intel's ICX compiler (version 2024.2.1, Build 20240711). Each optimization was compiled with the -fast flag, enabling aggressive optimizations to maximize performance.

2 Code Optimizations

2.1 Parallelize first loop

For the initial optimization, we parallelized the first for loop in the seq_kmeans function. This function also includes two other loops within the inlined functions euclid_dist and find_nearest_cluster. However, parallelizing these loops resulted in poorer performance due to the added overhead of thread creation and destruction, combined with their relatively low iteration count (only numCoords iterations in the euclid_dist function).

We then focused on the first outer loop within the do-while body. However, parallelizing this loop required addressing a few key challenges. Specifically, updates to the arrays newClusterSize and newClusters need to occur atomically, so we introduced a critical section in the code, ensuring that each thread updates these arrays sequentially. Additionally, the scalar variable delta requires safe concurrent updates. To avoid adding another critical section, we utilized a reduction approach. Each thread maintains its own copy of delta during execution, and, at the end of the parallel loop, a reduction operation aggregates these values, producing the final delta result by summing all thread contributions.

We apply the following changes to the code:

```
do {
    delta = 0.0;

    #pragma omp parallel for private(j, index) reduction(+:delta)
    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 */
        #pragma omp critical
        {
```

```
        newClusterSize[index]++;
        for (j=0; j<numCoords; j++)
            newClusters[index][j] += objects[i][j];
        }
    }
    //... second loop.
```

The above optimizations lead to a significant increase in performance as show in figure 1:

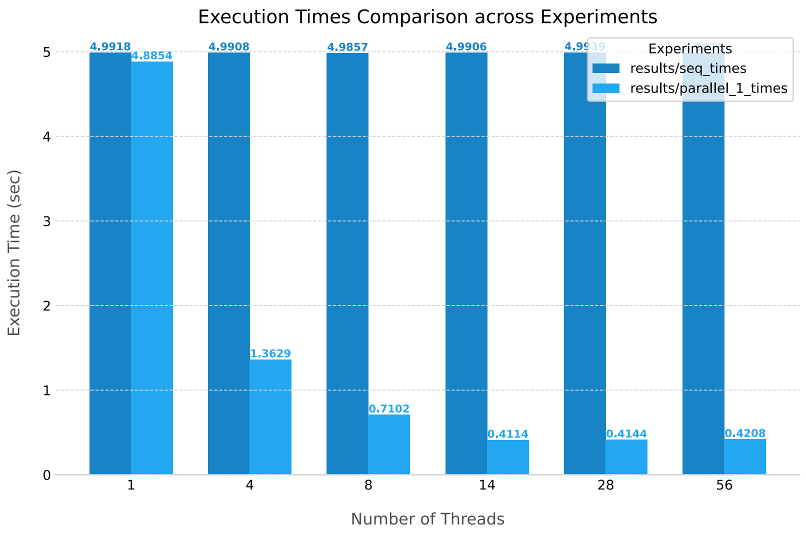

Figure 1: Comparison between sequential K-means and optimization #1

Here, we see a substantial performance boost when comparing the optimization with the original sequential k-means clustering algorithm. The performance boost is up to x12 for thread count of 14-56.

2.2 Parallelize second loop

The next logical optimization involves parallelizing the second **for** loop within the **do while** body. This loop averages the sums of the new clusters and updates the old cluster centers accordingly. Since each iteration is independent, this loop can be parallelized without restrictions.

Additionally, we separate the **omp parallel** directive from the **omp for** directive to include the second **for** loop within the same parallel region. This approach creates a single parallel region, thereby avoiding the overhead associated with creating and destroying threads multiple times.

The new code is shown below:

```
do {
    delta = 0.0;
        #pragma omp parallel
    {
        #pragma omp for private(j, index) reduction(+:delta)
        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 */
            #pragma omp critical
            {
            newClusterSize[index]++;
            for (j=0; j<numCoords; j++)
                newClusters[index][j] += objects[i][j];
            }
        }
        /* average the sum and replace old cluster center with newClusters */
        #pragma omp 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 */
        }
    }
    delta /= numObjs;
} while (delta > threshold && loop++ < 500);
```

With this optimization we get the following results as shown in figure 2

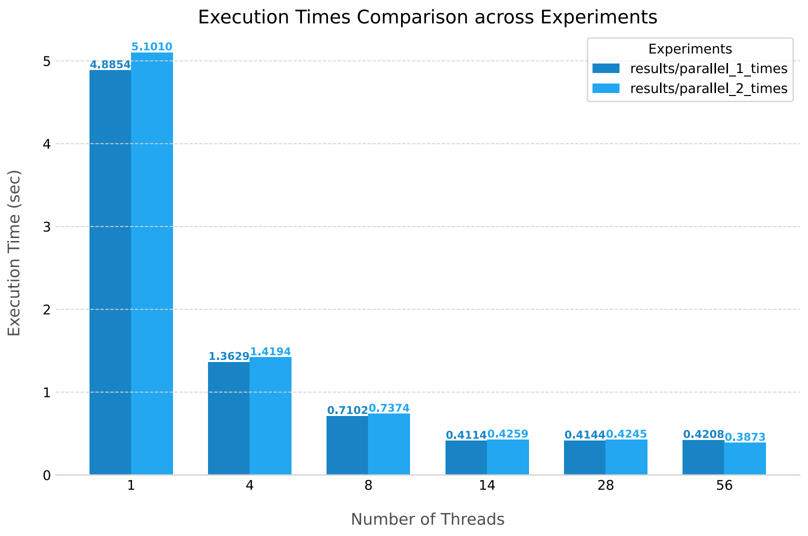

Figure 2: Comparison between optimization #1 and #2

The results indicate that, overall, the performance is not as good as the previous optimization, except for the case with 56 threads, which achieved the lowest execution time so far. Therefore, we will retain the second optimization as it offers the best performance improvements when utilizing a big number of threads.

2.3 Reduction on newClusterSize array

Our next optimization targets the first for loop by reducing the critical section, which currently adds significant overhead by restricting thread parallelism. To achieve this, we apply the reduction directive to the newClusterSize array, allowing each thread to maintain its own copy of the array. These copies are then combined to produce the final results, eliminating the need for a critical section. This approach reduces synchronization costs, as well as replaces the critical section with a single atomic operation for the newClusters computation:

```
do {
    delta = 0.0;
    #pragma omp parallel
    {
        #pragma omp for private(j, index) reduction(+:delta, newClusterSize[:numClusters])
        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]++;
            for (j=0; j<numCoords; j++) {
                #pragma omp atomic
                newClusters[index][j] += objects[i][j];
            }
        }
        //... second loop.
    }
```

As seen from Figure 3 the new optimization, as expected, leads to a significant performance boost especially as the number of threads increases.

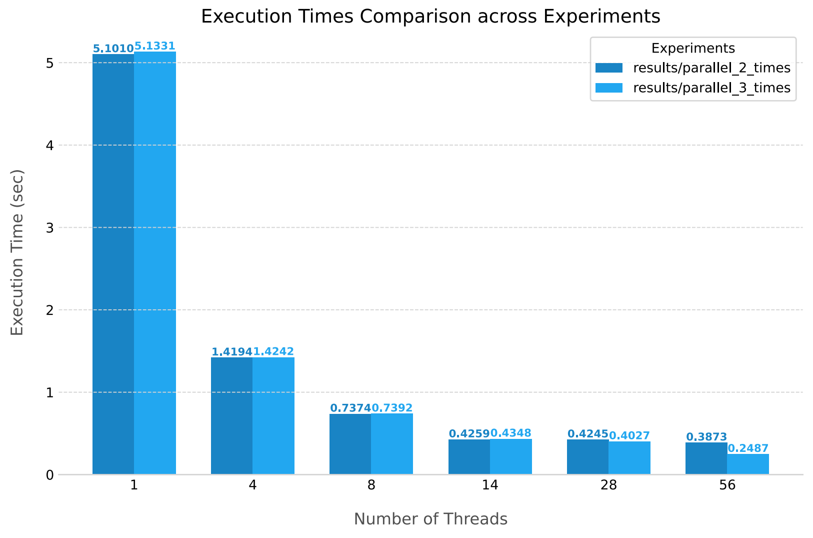

Figure 3: Comparison between optimization #2 and #3

2.4 Reduction on newClusters array

After reducing the critical section to a single atomic operation, the next logical step is to completely eliminate the atomic operation by introducing a fully parallelizable for loop. However, because the newClusters array is a non-contiguous 2D array, it cannot be directly reduced. To make this possible, we transform newClusters into a contiguous 1D array that supports reduction. By applying the reduce directive to this array, we achieve complete parallelization without any remaining atomic operations.

```
do {
    delta = 0.0;
    #pragma omp parallel
    {
        #pragma omp for private(j, index) reduction(+:delta, \
                                newClusterSize[:numClusters], \
                                newClusters[:numClusters*numCoords]\
                                )
        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]++;
            for (j=0; j<numCoords; j++) {
                newClusters[index*numCoords+j] += objects[i][j];
            }
        }
        //... second loop.
```

However, as shown in Figure 4, the new performance shows minimal improvement compared to the previous version.

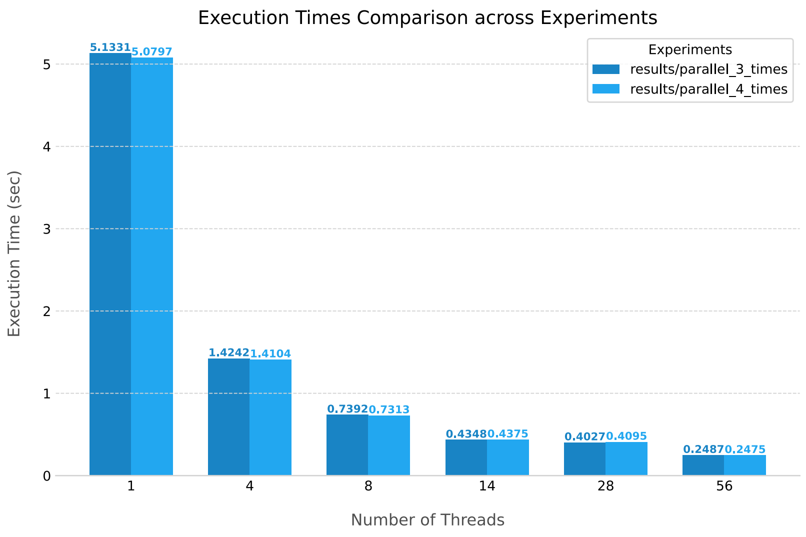

Figure 4: Comparison between optimizations #3 and #4

2.5 Parallel data structure initialization

With the two main loops in the seq_main function now fully parallelized, we next focus on parallelizing the initialization of the membership array at the start of the function. This initialization loop, which iterates over a large number of numObjs, is also fully parallelizable.

```
/* initialize membership[] */
#pragma omp parallel for
for (i=0; i<numObjs; i++) membership[i] = -1;
```

The results from this optimization however show no improvement. In contrast, we get slightly worse times due to the added overhead of a new parallel region inside the code.

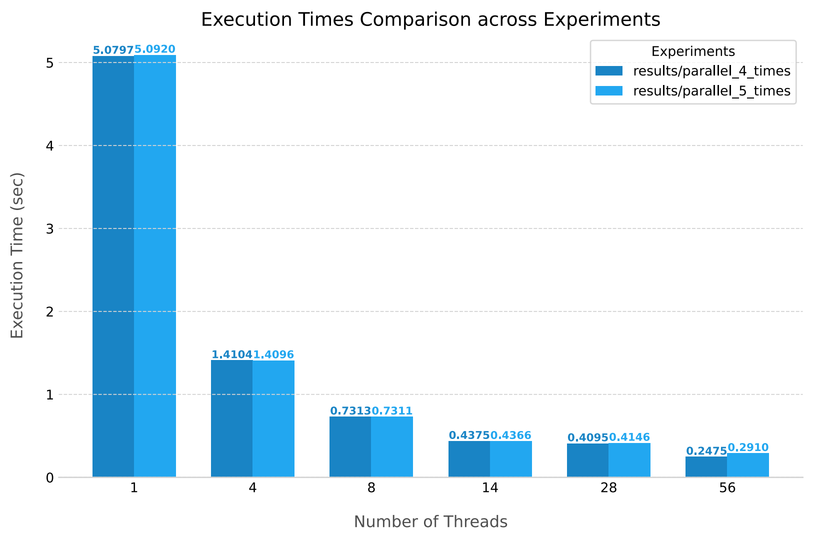

Figure 5: Comparison between optimizations #4 #5

2.6 OMP scheduling

At this point, all loops have been fully parallelized. Thus in this section, we experiment with different scheduling policies.

2.6.1 Custom scheduling

First, we apply a custom policy. We use dynamic scheduling for the first loop inside the do while body, and static scheduling for the second loop. We leave the chunk sizes to their default values.

```
do {
    delta = 0.0;
    #pragma omp parallel
    {
        #pragma omp for private(j, index) reduction(+:delta, \
                                    newClusterSize[:numClusters], \
                                    newClusters[:numClusters*numCoords]\
                                    schedule(dynamic))
        for (i=0; i<numObjs; i++) {
                ......
        }
        /* average the sum and replace old cluster center with newClusters */
        #pragma omp for private(j) \
                    schedule(static)
        for (i=0; i<numClusters; i++) {
                ......
        }
```

With the combination of these two policies we get the following results:

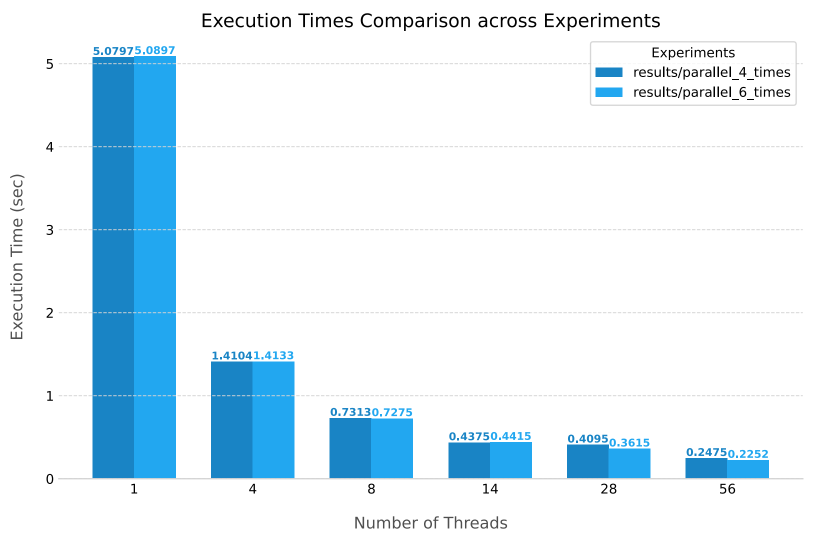

Figure 6: Comparison between optimizations #5 and #6

From Figure 6, we observe that we get the lowest execution time so far.

We then experimented with different chunk sizes using the same scheduling policies. For the dynamically scheduled loop, testing revealed that a chunk size of 4 yielded the best performance. Increasing the chunk size beyond 4 resulted in execution times worse than the default setting. Although chunk size 4 provided the optimal performance, the improvement over the default was not particularly substantial.

As for the statically scheduled loop, the default chunk size proved to deliver the best results.

Figure 7 shows the final execution times after experimenting with various scheduling policies. The times displayed in the graph reflect the algorithm's performance using dynamic scheduling with a chunk size of 4 for the first loop and static scheduling for the second loop.

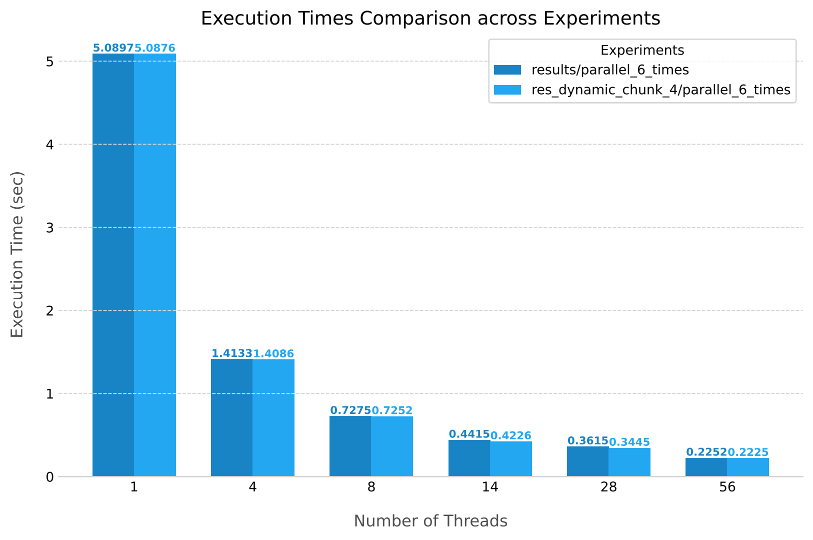

Figure 7: Comparison between optimizations #7 and #8

2.6.2 Auto scheduling

In addition to our chosen scheduling policy, we also use the auto option in the schedule directive, allowing the compiler to determine the optimal way to distribute iterations across threads. This approach enables the compiler to analyze the workload and select a scheduling strategy that best suits the loop characteristics and system architecture.

```
do {
    delta = 0.0;
    #pragma omp parallel
    {
        #pragma omp for private(j, index) reduction(+:delta, \
                                    newClusterSize[:numClusters], \
                                    newClusters[:numClusters*numCoords]\
                                    schedule(auto))
        for (i=0; i<numObjs; i++) {
                ......
        }
        /* average the sum and replace old cluster center with newClusters */
        #pragma omp for private(j) \
                    schedule(auto)
        for (i=0; i<numClusters; i++) {
                ......
        }
```

In Figure 8 we see the results from the auto scheduling policy.

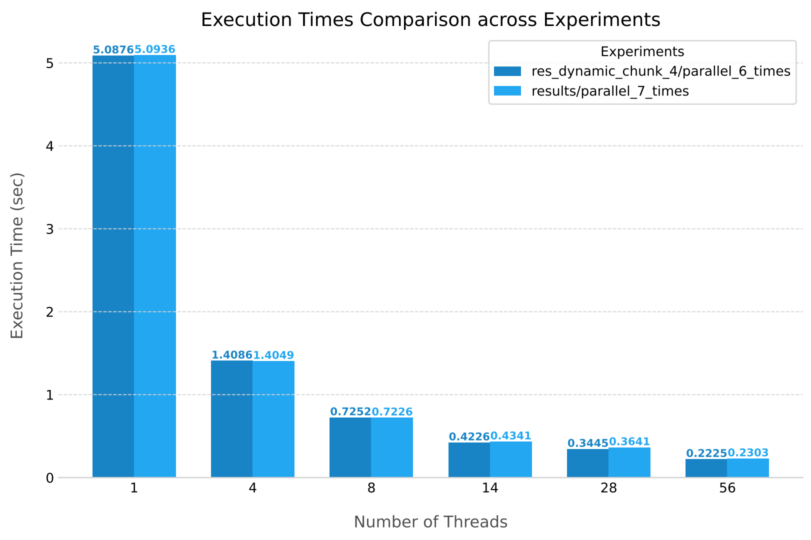

Figure 8: Comparison between optimizations #7 and #8

We find that the execution times are nearly identical to those from previous optimizations. However, with 56 threads, the performance still falls short of the minimum time achieved in optimization 6. As a result, we decide to discard the auto scheduling and retain our previous scheduling approach.

2.7 Streaming SIMD Extensions (SSE)

After fully exploiting parallelism in the main loops of the seq_kmeans function, we shift our focus to additional code optimizations. Intel VTune Profiler has identified that the majority of our program's runtime is spent within the Euclidean distance calculation. Although, as discussed in Section 2.1, parallelizing this loop resulted in degraded performance, there are still opportunities to optimize this critical section. One such approach involves utilizing Intel's Streaming SIMD Extensions (SSE) to accelerate the Euclidean distance computation.

```
float euclid_dist_2(int    numdims,  /* no. dimensions */
                    float *coord1,   /* [numdims] */
                    float *coord2)   /* [numdims] */
{
    int i;
    float ans=0.0;

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

    return(ans);
}
```

The original Euclidean distance function performs the following calculation:

$$dist = \sum_{i=0}^{numDims} (coord1[i] - coord2[i])^2 \qquad (1)$$

It calculates the sum of square difference for each coordinate i, one at a time. By leveraging SSE we can extend this summation and calculate the square root difference of distance for four coordinates at a time.

```
float euclid_dist_2(int numdims, float *restrict coord1, float *restrict coord2) {
    float ans = 0.0f;
    int i;

    // Initialize SIMD sum to 0
    __m128 sum = _mm_setzero_ps();

    for (i = 0; i <= numdims - 4; i += 4) {
        __m128 vec1 = _mm_loadu_ps(&coord1[i]);
        __m128 vec2 = _mm_loadu_ps(&coord2[i]);
        __m128 diff = _mm_sub_ps(vec1, vec2);
        sum = _mm_add_ps(sum, _mm_mul_ps(diff, diff));
    }

    // Perform horizontal addition on sum
    float tmp[4];
    _mm_storeu_ps(tmp, sum);
    ans = tmp[0] + tmp[1] + tmp[2] + tmp[3];

    // Process remaining elements
    for (; i < numdims; i++) {
        float diff = coord1[i] - coord2[i];
        ans += diff * diff;
    }

    return ans;
}
```

Using Intel's intrinsics library, we define a 128-bit vector capable of holding four 32-bit floats. We then calculate the square root difference using this vector.

With this optimization, we get as expected, a significant performance boost when compared with the previous ones. Figure 9 shows the new results.

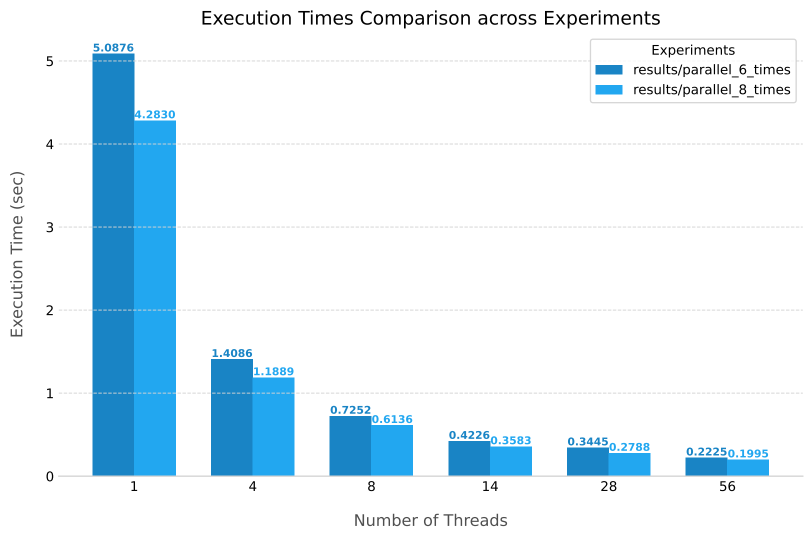

Figure 9: Comparison between optimization #7 and #8

2.8 Advanced Vector Extensions (AVX)

For our last optimization, we went one step further and implemented the euclidean function using AVX, introducing 256-bit registers that can hold and perform operations on eight 32-bit floating point values at a time. The final code is the same as the one in the previous optimization with the only difference of using the __m256 data type and its respective functions for addition, subtruction and multiplication.

However, as seen in Figure 10, this optimization had slightly worse times compared to the previous ones. The extra overhead of handling twice the data had a negative effect on performance, possibly due to the low iteration count of the euclidean distance loop (two iterations on our test where we used 22 dimensions for coordinates).

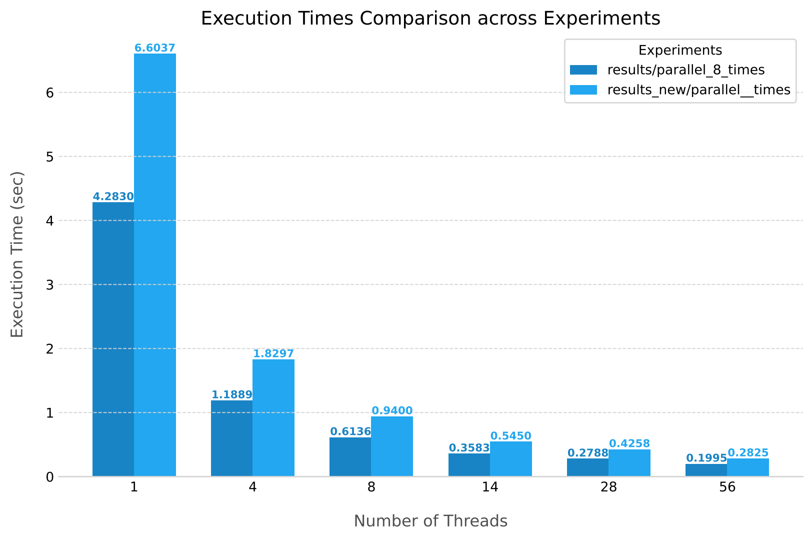

Figure 10: Comparison between optimization #8 and #9

3 Conclusions

In this project, we explored several strategies, including parallelizing the main loops, implementing reduction techniques to minimize synchronization overhead, and experimenting with different scheduling policies. Our findings indicated that while some optimizations yielded substantial performance gains, others had negligible effects or even degraded performance due to overhead. Furthermore, we learned that aggressive parallelization does not always lead to the best performance, as it can introduce additional overhead and complexity since the effectiveness of each optimization is also influenced by various factors such as the size of the dataset or the nature of the algorithm.

Figure 11 illustrates the mean execution time of all optimizations at the thread count that achieved the minimum execution times, which in our case is 56 threads.

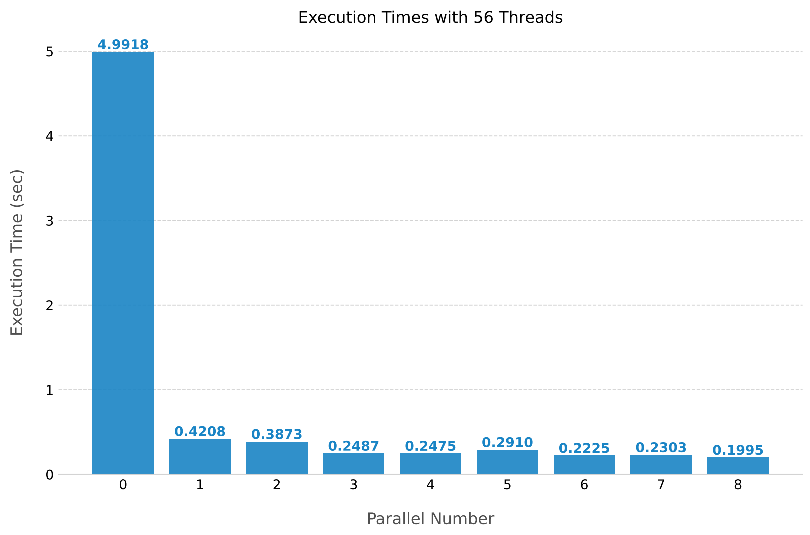

Figure 11: Final execution times for all the optimizations

After implementing a series of optimizations, we achieved a performance speedup of 25 times compared to the original sequential algorithm. Each optimization contributed significantly to this enhancement, with individual improvements ranging from 12 times to as much as 25 times over the initial implementation.