Department of Electrical and Computer Engineering

University of Thessaly

ECE415: High Performance Computing (HPC)

Lab 2: Parallel Implementation of K-Means clustering algorithm using OpenMP

Ioanna-Maria Panagou, 2962

Nikolaos Chatzivangelis, 2881

Fall Semester 2022-2023

Contents

1 Summary

In this lab, we used OpenMP directives to parallelize a sequential version of the K-means clustering algorithm. Our quickest implementation on csl-artemis used 64 threads to achieve a speedup of **x27.5**.

2 Introduction

The K-means algorithm is used to partition `numObjs` objects consisting of `numCoords` coordinates into `numClusters`. Our algorithm is comprised of the following steps:

1. Read the points to be clustered from an input file.
2. Use the first `numClusters` points from the input file to initialize the cluster centers.
3. Iterate over the `numObjs` objects and compute its euclidian distance from all the cluster centers.
4. Assign the membership of each object to the cluster that is closer in distance from it.
5. If the membership of the object changed, increment a variable delta that is used to determining if the algorithm has converged.
6. Update the new cluster logistics (size and coordinates).
7. If the variable delta is more than a predetermined threshold or the maximum number of repetitions has not been reached, proceed to step 3.
8. Update the clusters coordinates as the average of the points that belong to each cluster.

The remainder of our report is structured as follows: section 3 describes the process we followed to evaluate our optimizations, 4 presents our proposed optimizations steps and section 5 concludes the report.

3 Evaluation

After using VTune profiler to profile our code, we identify the for loops that account for the largest percentages of execution time and attempt to parallelize them one at a time in descending order of importance using OpenMP pragmas. If our attempt proved unsuccessful in reducing the execution time, we discard it and move on with the next loop. We run each implementation 12 times for 1, 4, 8, 16, 32 and 64 threads on csl-artemis (2 16-way multicore CPUs that support 2-way hyperthreading) and compute the average execution time and standard deviation excluding the mimimum and maximum measurements.

4 Optimization Steps

For reference, the mean execution time for the baseline sequential implementation was 5.485 ± 0.009s. We compiled with `OPTFLAGS=-qopenmp -fast -DNDEBUG -g` and the icx compiler.

4.1 Optimization step #1

The VTune profiler hinted that the loop in the `find_nearest_cluster` function was taking the majority of the execution time, so we decided to parallelize it. This for loop iterates over all clusters and returns the index of the cluster with the minimum euclidean distance from the object that was given as an input parameter to this function.

```
1  index = 0;
2  min_dist = euclid_dist_2(numCoords, object, clusters[0]);
3
4  for (i=1; i<numClusters; i++) {
5      dist = euclid_dist_2(numCoords, object, clusters[i]);
6      /* no need square root */
7      if (dist < min_dist) { /* find the min and its array index */
8          min_dist = dist;
9          index    = i;
10     }
11 }
12 return(index);
```

Listing 1: Original code

The `dist` variable would have had to be declared as private and the `min_dist` as shared, thus it would be necessary to enclose the code in lines 7-10 from listing 3 in a critical section. As we know, critical sections can negatively affect the performance of our code, since synchronization is needed in order for the threads to execute the section one at a time and threads must wait for their turn to enter the critical section, thus making the code essentially sequential. To avoid having to add a `# pragma omp critical` pragma, we created a new data type as a struct that contains an index and a distance that replaced the `min_dist` and `index` variables. Then, we declared our own reduction using the `# pragma omp declare reduction` directive on this data type that returns the struct with the minimum distance from the input object.

```
1  typedef struct dist_t {
2      double dist;
3      int index;
4  } dist_t;
5
6  #pragma omp declare reduction (min_dist_reduction : dist_t : omp_out = ( (
       omp_in.dist < omp_out.dist)  \
7   || ( (omp_in.dist == omp_out.dist ) && (omp_in.index < omp_out.index ) ))?
       omp_in : omp_out ) \
```

```
8  initializer(omp_priv = {DBL_MAX, 0})
9
10 dist_t min_dist = {DBL_MAX, 0};
11
12     /* find the cluster id that has min distance to object */
13
14 #pragma omp parallel  private(dist), reduction(min_dist_reduction:min_dist)
15 for (i=0; i<numClusters; i++) {
16     dist = euclid_dist_2(numCoords, object, clusters[i]);
17     /* no need square root */
18     if (dist < min_dist.dist) { /* find the min and its array index */
19         min_dist.dist = dist;
20         min_dist.index  = i;
21     }
22
23 }
24 return(min_dist.index);
```

Listing 2: Parallel Code

Our custom reduction compares the distance between two `dist_t` structs and returns the ones with the minimum distance, or, in case of a draw, the one with the smallest index. Each thread now keeps a local copy of the `min_dist` variable initialized with {DBL_MAX, 0}. Notice that we need an initializer in line 8, because, once the threads enter the parallel region, we cannot be sure about the initial values of their copy of the `min_dist` variable. Also, the loop indexing variable now starts from 0 instead of 1.

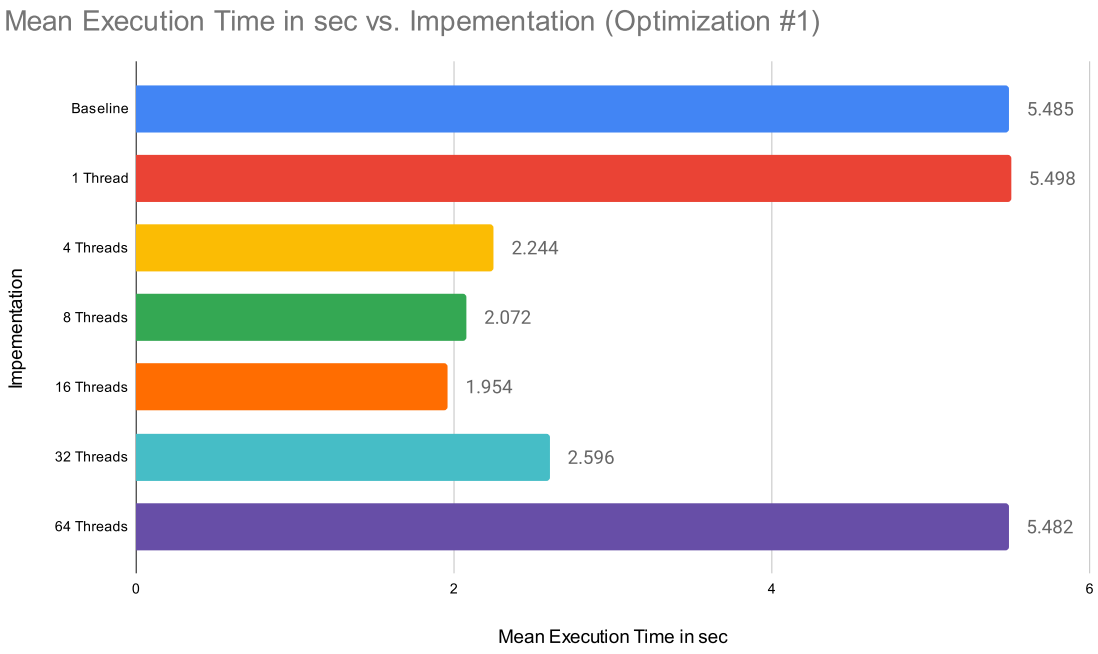

Figure 1: Mean Execution Time in sec for Optimization #1

We managed to reach an execution time of 1.954 ± 0.051s using 16 threads on csl-artemis, reaching a speedup of x2.8. As expected, the execution time of the parallel implementation using only 1 thread was slightly greater than the baseline, since we now have the overhead of creating and destroying that particular thread. In addition, increasing the number of threads beyond a certain threshold has a negative impact on the execution time.

4.2 Optimization step #2

The next loop that was highlighted by Vtune was the loop in the euclidean distance function. We noticed that this loop iterates over the coordinates of each point, which in our case are 20. Obviously, creating 32 or 64 threads to handle 20 iterations would be detrimental to performance. Additionally, the `euclidean_distance` function is called iteratively in a loop, so each time we would have the creation and destruction of threads, which would add significant overhead. We attempted to parallelize this loop, but it actually doubled the execution time, so we decided to move outwards and parallelize the loop in the `seq_kmeans` function that iterates over the objects. In this for loop, the `find_nearest_cluster` function that we optimized in the previous step is repeatedly called. By default, nested parallelism is off, so the thread that encounters the parallel construct in the `find_nearest_cluster` will just create a team of threads that consists only of itself, meaning additional overhead. If we decide to enable nested parallelism (export OMP_NESTED=TRUE), then the encountering thread will pick some idle threads from a pool of available threads. However, the number of clusters are much more than the available CPU threads and parallelizing the inner loop (the loop from the previous step) will not have a positive effect on the execution time, since we have used all CPUs on the outer loop. For that reason, we remove the omp pragma from optimization step 1 and parallelize the first loop in the `seq_kmeans` function.

```
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) 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][j] += objects[i][j];
}
```

Listing 3: Original code

```
#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 atomic update
    newClusterSize[index]++;
    for (j=0; j<numCoords; j++)
        #pragma omp atomic update
        newClusters[index][j] += objects[i][j];
}
```

Listing 4: Parallel Code

We managed to reach an execution time of 0.247 ± 0.017s using 64 threads on csl-artemis, reaching a speedup of x22.

The variables `index` and `j` should be declared as private and we could declare `delta` as the subject to a reduction operator, since it is constantly incremented by 1. The update to the `newClusterSize` and `newClusters` should be performed by only one thread at a time to avoid race conditions, since the variable `index` is private to each thread and in one iteration two threads may update the same entry. We preferred the `# pragma omp atomic update` pragmas instead of two critical regions, because they can have lower overhead than critical sections if HW atomics are leveraged.

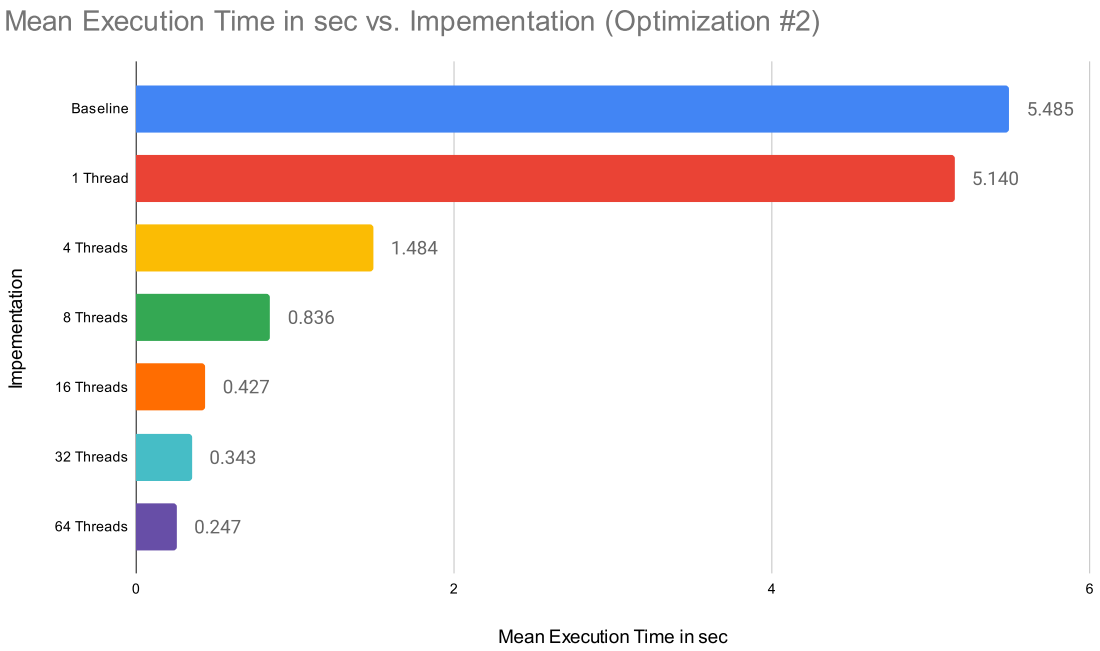

Figure 2: Mean Execution Time in sec for Optimization #2

4.3 Optimization step #3

In order to get the most out of the treads we created in the previous optimization step, we decided to extend the parallel region to enclose both loop in the `seq_kmeans` function.

```
#pragma omp parellel 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 atomic update
    newClusterSize[index]++;
    for (j=0; j<numCoords; j++)
        #pragma omp atomic update
        newClusters[index][j] += objects[i][j];
}

/* average the sum and replace old cluster center with newClusters */
```

```
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 */
}
```

Listing 5: Optimization step #2

```
#pragma omp parallel
{
#pragma omp for private(j, index), reduction(+:delta)
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) delta += 1.0;

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

    /* update new cluster center : sum of objects located within */
    #pragma omp atomic update
    newClusterSize[index]++;
    for (j=0; j<numCoords; j++)
        #pragma omp atomic update
        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;
}
```

Listing 6: Optimization step #3

Interestingly enough, although on average it performed better than optimization #2, it is slower when using 64 threads, which is the fastest implementation, so we did not include it in the rest of our optimizations.

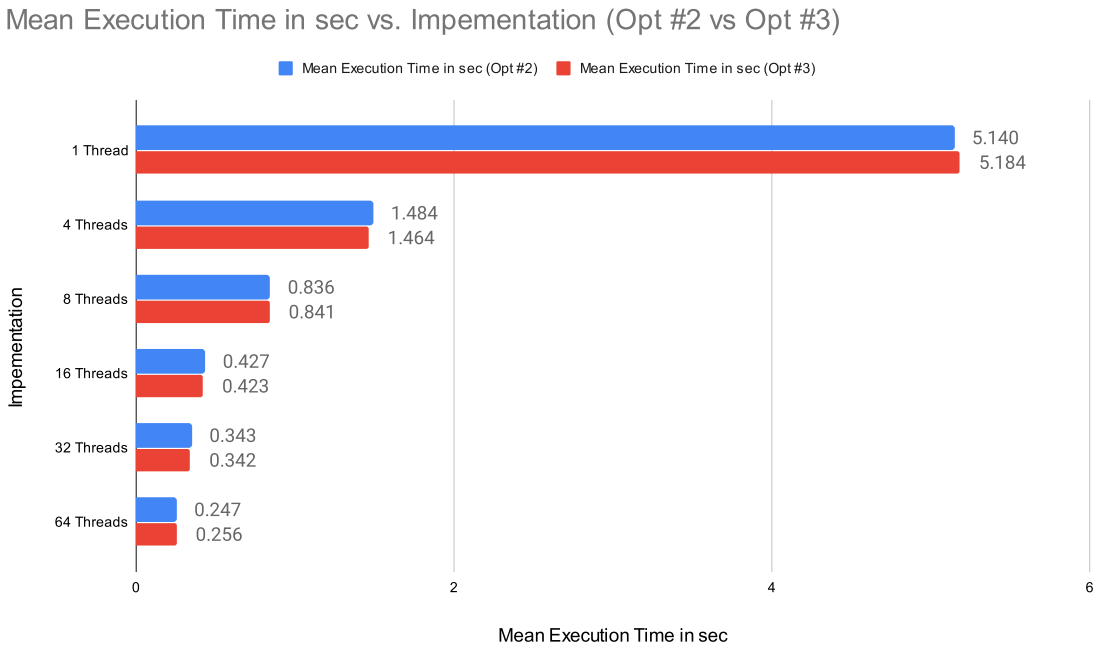

Figure 3: Mean Execution Time in sec for Optimization #2 and Optimization #3

4.4 Optimization step #4

In an attempt to further reduce the execution time, we parallelized the loop that initializes memberships, even though it is performed only once during a run.

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

Listing 7: Optimization step #2

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

Listing 8: Optimization step #4

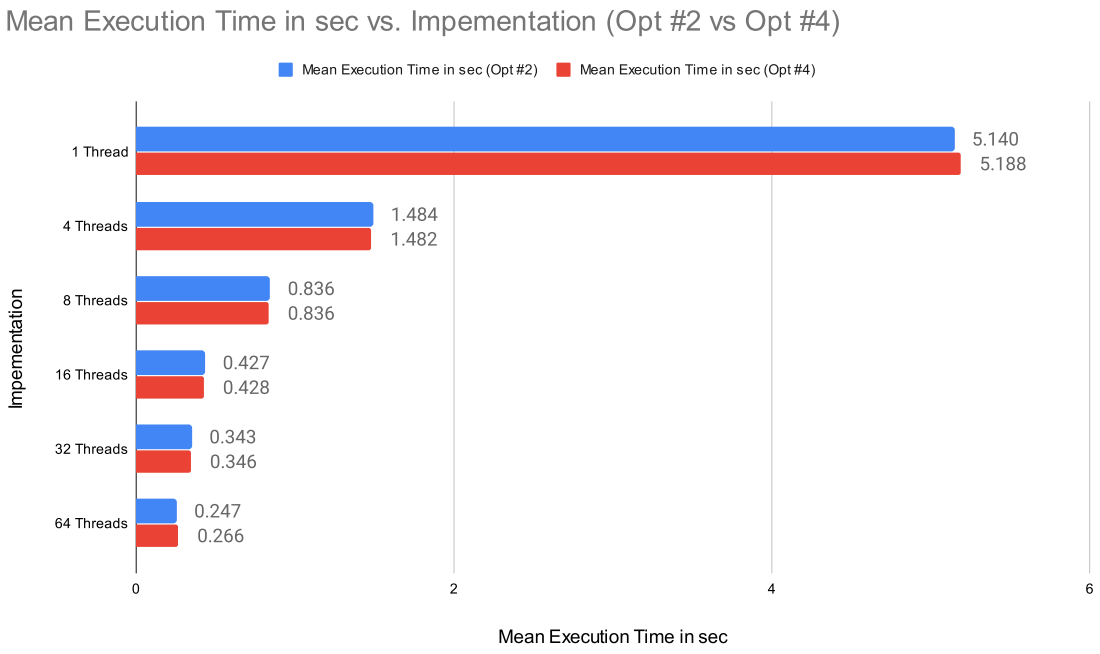

Figure 4: Mean Execution Time in sec for Optimization #2 and Optimization #4

As we can see from fig.8, this version is on average slower than the previous one, which implies that the execution time of this particular loop when executed sequentially, is so small that the added parallelization adds significant overhead from creating and destroying threads.

4.5 Optimization step #5

In order to mitigate the added overhead of the parallelization of the previous step, we extended the parallel region to cover both initialization loops - the one that initializes memberships and the newClusters.

```
#pragma omp parallel for
for (i=0; i<numObjs; i++)
    membership[i] = -1;
for (i=1; i<numClusters; i++)
    newClusters[i] = newClusters[i-1] + numCoords;
```

Listing 9: Optimization step #4

```
#pragma omp parallel
{
#pragma omp for nowait
for (i=0; i<numObjs; i++)
    membership[i] = -1;
#pragma omp for
for (i=1; i<numClusters; i++)
    newClusters[i] = newClusters[0] + i*numCoords;
```

```
9 }
```

Listing 10: Optimization step #5

The two loops in the parallel region update different memory locations, so there is no need for the implicit barrier after the first loop and, thus, we removed it with nowait. The second loop needed some manipulation in order to be parallelizable. We could think of it as a recurrent relation $T(n) = T(n-1) + numCoords$. We will try to solve this recurrent relation as:

$$\begin{aligned} T(i) &= T(i-1) + numCoords \\ &= T(i-2) + 2 * numCoords \\ &= T(i-3) + 3 * numCoords \\ &= T(i-n) + n * numCoords \qquad \text{(we substitute n with i)} \\ &= T(0) + i * numCoords \end{aligned}$$

, where $T(i)$ is equal to `newClusters[i]`

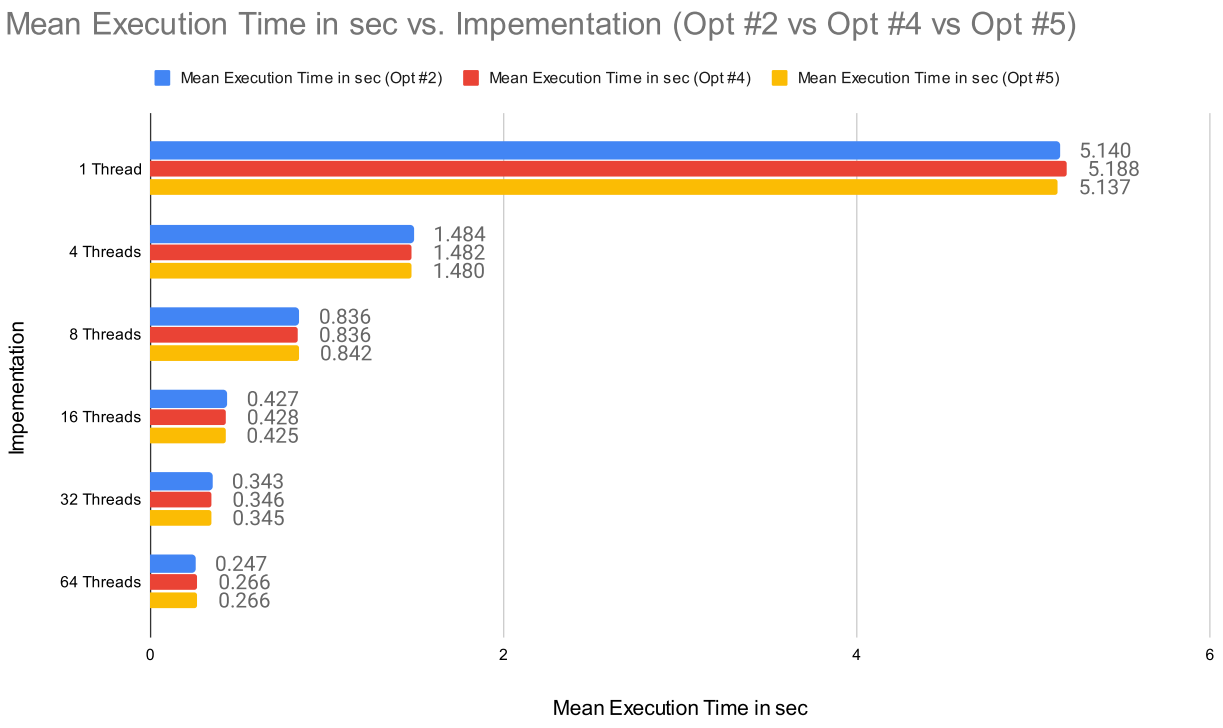

Figure 5: Mean Execution Time in sec for Optimizations #2, #4 an #5

Again, this optimization did not make a significant impact and was slower for the fastest implementation using 64 threads, so we abandoned the parallelization of the two initialization loops all together.

4.6 Optimization step #6

Even though the load is more or less balanced among the different threads in optimization #2, we still tried to change the schedule to dynamic to test if we can gain some additional speedup. Our initial thought, however, proved to be correct and the dynamic schedule increased the execution time (in the case of 64 threads) for all chunks we tried as seen in fig.6

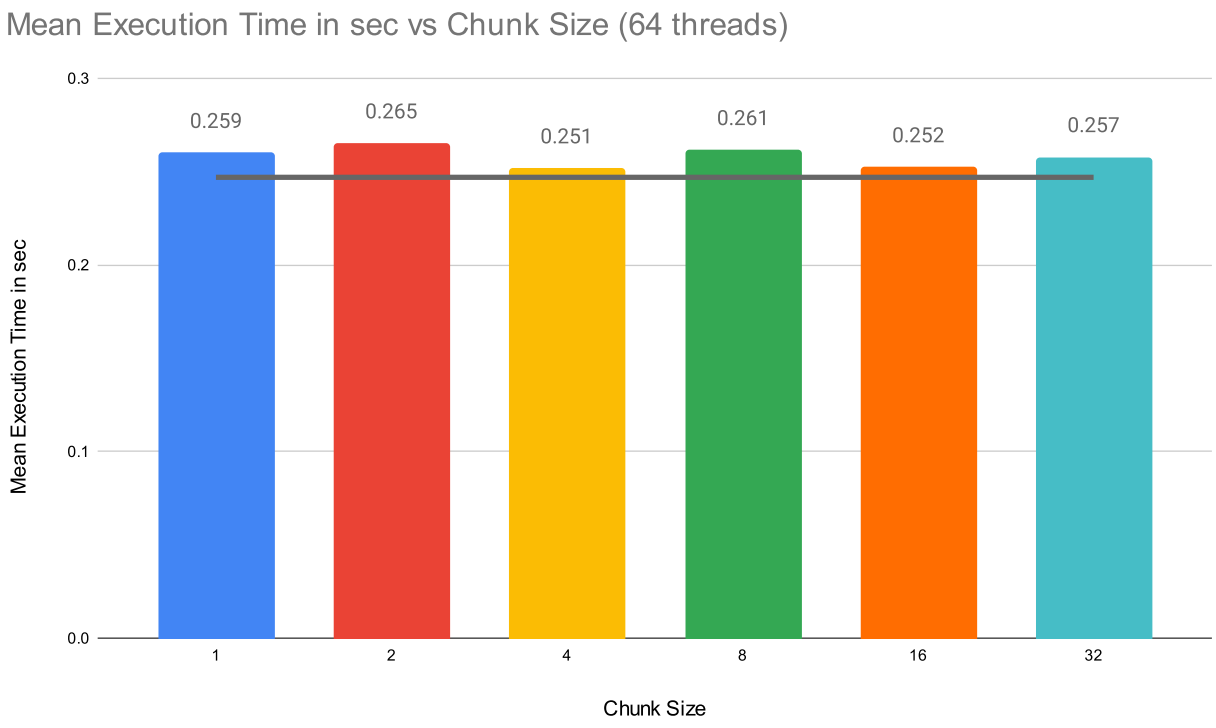

Figure 6: Mean Execution Time in sec for Optimization step #6 (64 threads) for different chunk sizes. The grey horizontal line is the corresponding mean execution time for Optimization step #2 using 64 threads

4.7 Optimization Step #7

Looking back at the loop in optimization step #2, we understood that we could perform a reduction on the `newClusterSize` array (line 15 of 11) and get rid of the atomic pragma that slows down our code.

```
1 #pragma omp parallel for private(j, index), reduction(+:delta), schedule(
    auto)
2 for (i=0; i<numObjs; i++) {
3     /* find the array index of nestest cluster center */
4     index = find_nearest_cluster(numClusters, numCoords, objects[i],
5                                  clusters);
6
7     /* if membership changes, increase delta by 1 */
8     if (membership[i] != index) delta += 1.0;
```

```
9
10      /* assign the membership to object i */
11      membership[i] = index;
12
13      /* update new cluster center : sum of objects located within */
14      #pragma omp atomic update
15      newClusterSize[index]++;
16      for (j=0; j<numCoords; j++)
17          #pragma omp atomic update
18          newClusters[index][j] += objects[i][j];
19  }
```

Listing 11: Optimization step #2

```
1  #pragma omp parallel for private(j, index),  \
2   reduction(+:delta, newClusterSize[:numClusters]), schedule(auto)
3  for (i=0; i<numObjs; i++) {
4      /* find the array index of nestest cluster center */
5      index = find_nearest_cluster(numClusters, numCoords, objects[i],
6                                   clusters);
7
8      /* if membership changes, increase delta by 1 */
9      if (membership[i] != index) delta += 1.0;
10
11     /* assign the membership to object i */
12     membership[i] = index;
13
14     /* update new cluster center : sum of objects located within */
15     newClusterSize[index]++;
16     for (j=0; j<numCoords; j++)
17         #pragma omp atomic update
18         newClusters[index][j] += objects[i][j];
19 }
20
21 /* average the sum and replace old cluster center with newClusters */
22 for (i=0; i<numClusters; i++) {
23     for (j=0; j<numCoords; j++) {
24         if (newClusterSize[i] > 0)
25             clusters[i][j] = newClusters[i][j] / newClusterSize[i];
26         newClusters[i][j] = 0.0;   /* set back to 0 */
27     }
28     newClusterSize[i] = 0;   /* set back to 0 */
29 }
```

Listing 12: Optimization step #7

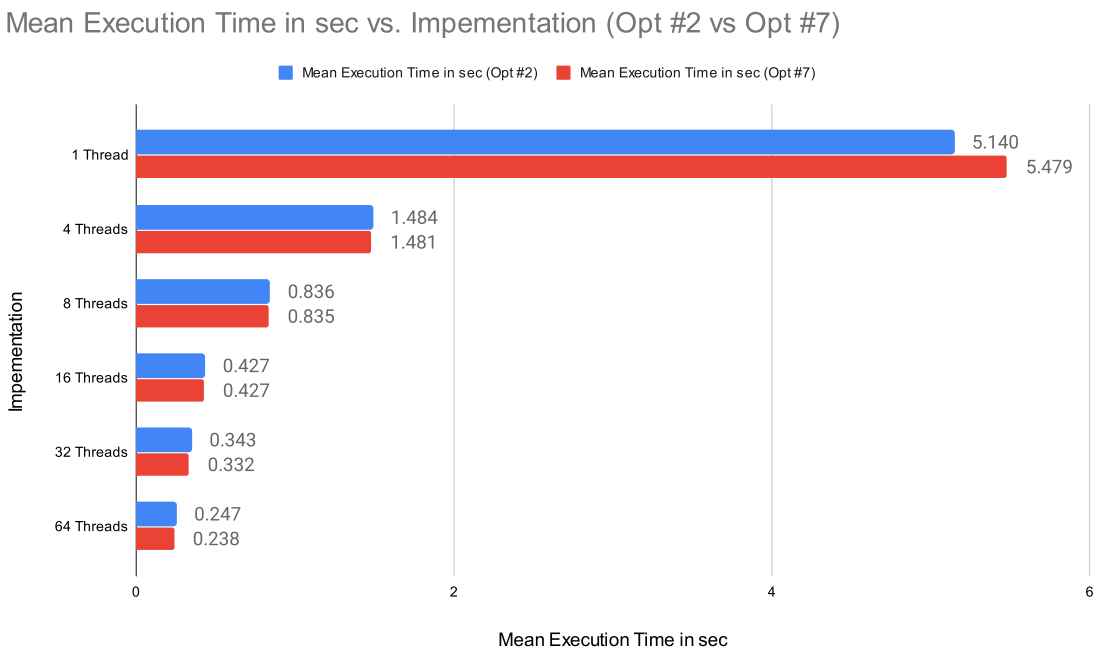

Figure 7: Mean Execution Time in sec for Optimization #2 and Optimization #7

As seen in fig.7, this was the first implementation after #2 that positively affected the execution time.

4.8 Optimization step #8

Since the previous optimization step proved to be effective, we decided to eliminate the second atomic pragma in line 17 of listing 12. In order to do so, we had to transform the `newClusters` data structure into a one dimensional array as presented in listings 13 and 14. By converting the `newClusters` structure into a one dimensional dynamically allocated array, we can avoid the loop in lines 5-7 of 13, but we also need to change the indexing (`newClusters[index][j]` to `newClusters[index * numCoords + j]`)

```
1  newClusters    = (float**) malloc(numClusters *            sizeof(float*));
2  assert(newClusters != NULL);
3  newClusters[0] = (float*)  calloc(numClusters * numCoords, sizeof(float));
4  assert(newClusters[0] != NULL);
5  for (i=1; i<numClusters; i++)
6      newClusters[i] = newClusters[i-1] + numCoords;
7
8  #pragma omp parallel for private(j, index), reduction(+:delta,
       newClusterSize[:numClusters]), , schedule(auto)
9  for (i=0; i<numObjs; i++) {
10     /* find the array index of nestest cluster center */
11     index = find_nearest_cluster(numClusters, numCoords, objects[i],
12                                  clusters);
```

```
13
14      /* if membership changes, increase delta by 1 */
15      if (membership[i] != index) delta += 1.0;
16
17      /* assign the membership to object i */
18      membership[i] = index;
19
20      /* update new cluster center : sum of objects located within */
21      newClusterSize[index]++;
22      for (j=0; j<numCoords; j++)
23          #pragma omp atomic update
24          newClusters[index][j] += objects[i][j];
25  }
```

Listing 13: Before optimization

```
1  newClusters = (float*)calloc(numClusters*numCoords, sizeof(float));
2  assert(newClusterSize != NULL);
3
4  ...
5
6  #pragma omp parallel for private(j, index), reduction(+:delta, \
       newClusterSize[:numClusters],newClusters[:numClusters*numCoords]),
       schedule(auto)
7  for (i=0; i<numObjs; i++) {
8      /* find the array index of nestest cluster center */
9      index = find_nearest_cluster(numClusters, numCoords, objects[i],
10                                  clusters);
11
12     /* if membership changes, increase delta by 1 */
13     if (membership[i] != index) delta += 1.0;
14
15     /* assign the membership to object i */
16     membership[i] = index;
17
18     /* update new cluster center : sum of objects located within */
19     newClusterSize[index]++;
20     for (j=0; j<numCoords; j++)
21         newClusters[index * numCoords + j] += objects[i][j];
22 }
```

Listing 14: After Optimization

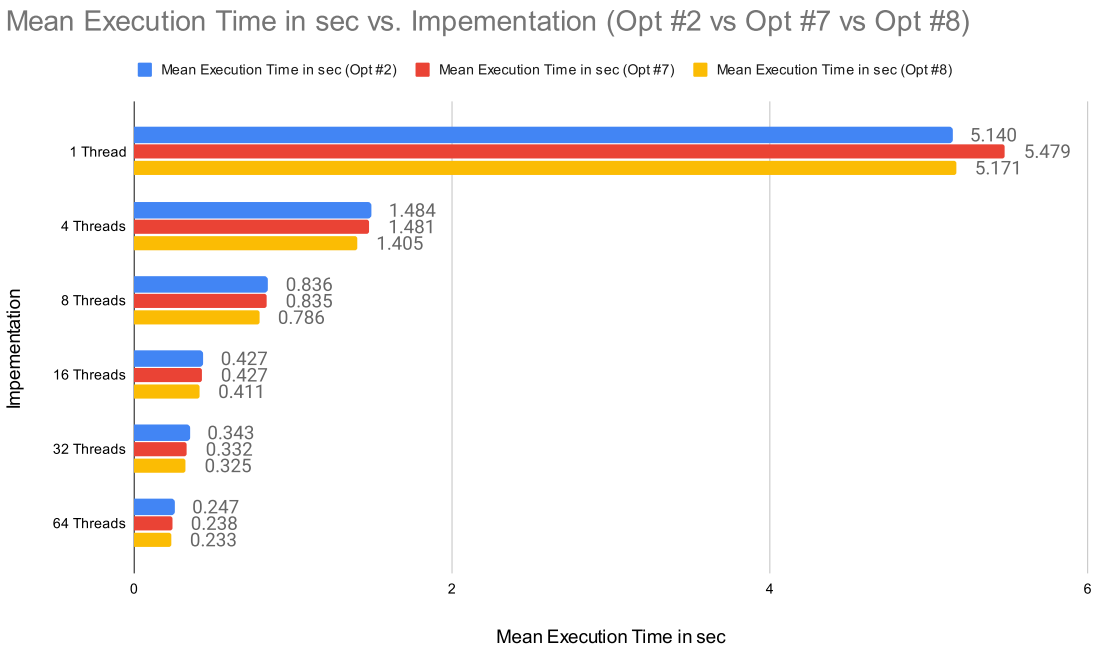

Figure 8: Mean Execution Time in sec for Optimizations #2, #7 and #8

4.9 Optimization step #9

As our last optimization, we shifted our attention towards the loop in the `euclidean_dist` function, which we unsuccessfully tried to parallelize earlier. This loop is the most time consuming portion of our code, because it involves `numCoords` multiplications. Since parallelization turned out to have a negative impact on the execution time, we figured that we could execute many operations concurrently, utilizing the SIMD units.

```
__inline static
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);
}
```

Listing 15: Before optimization

```
__forceinline float hadd_ps( __m128 r4 )
{
```

```
3
4   /* r2 is the result of the addition of the input vector and the input
      vector
5    * with the 2 lower and 2 higher floats interchanged */
6   const __m128 r2 = _mm_add_ps( r4, _mm_movehl_ps( r4, r4 ) );
7
8   /* r1 is the result of the addition of r2 and r2 with the floats in
9    * in odd indexed duplicated */
10  const __m128 r1 = _mm_add_ss( r2, _mm_movehdup_ps( r2 ) );
11
12  /* the  result is in float 0 of vector r1 */
13  return _mm_cvtss_f32( r1 );
14 }
15
16 __inline static
17 float euclid_dist_2(int    numdims,  /* no. dimensions */
18                     const float *coord1,   /* [numdims] */
19                     const float *coord2)   /* [numdims] */
20 {
21     int i;
22     float ans=0.0;
23     const float *p1 = coord1;
24     const float *p2 = coord2;
25     const float *const p1End = p1 + numdims;
26     __m128 acc;
27
28     const __m128 a = _mm_load_ps(p1);
29     const __m128 b = _mm_load_ps(p2);
30     const __m128 c = _mm_sub_ps(a, b);
31     acc = _mm_mul_ps(c, c);
32
33     p1 += 4;
34     p2 += 4;
35
36     for (; p1 < p1End; p1+=4, p2+=4) {
37       const __m128 a = _mm_load_ps(p1);
38       const __m128 b = _mm_load_ps(p2);
39       const __m128 c = _mm_sub_ps(a, b);
40       acc = _mm_fmadd_ps(c, c, acc);
41     }
42
43     ans = hadd_ps(acc);
44     return(ans);
45 }
```

Listing 16: After Optimization

We created 3 SIMD vectors:

- Vector `a` contains 4 elements from the `coord1` array in indices 0-3 or 4-7 etc.
- Similarly, vector `b` contains 4 elements from the `coord2` array
- Vector `c` is the difference `coord1 - coord2`
- In vector `acc` we stored the result of a fused-multiply add operation, i.e $c * c + acc$

We should not that this implementation is only valid when the number of coordinate dimensions is a multiple of 4. After the loop is through, our result is the sum of all the elements of the vector `acc`. We created a new horizontal add operation that is comprised of the below steps:

1. The `_mm_movehl_ps` function takes as parameters two SIMD 128-bit vectors a and b and returns a new 128-bit vector that has as its two lower elements the 2 higher elements of b and as its two higher elements the 2 lower elements of a. In our case, a and b are the same vector (the acc vector) and a new vector is returned as shown in fig.9

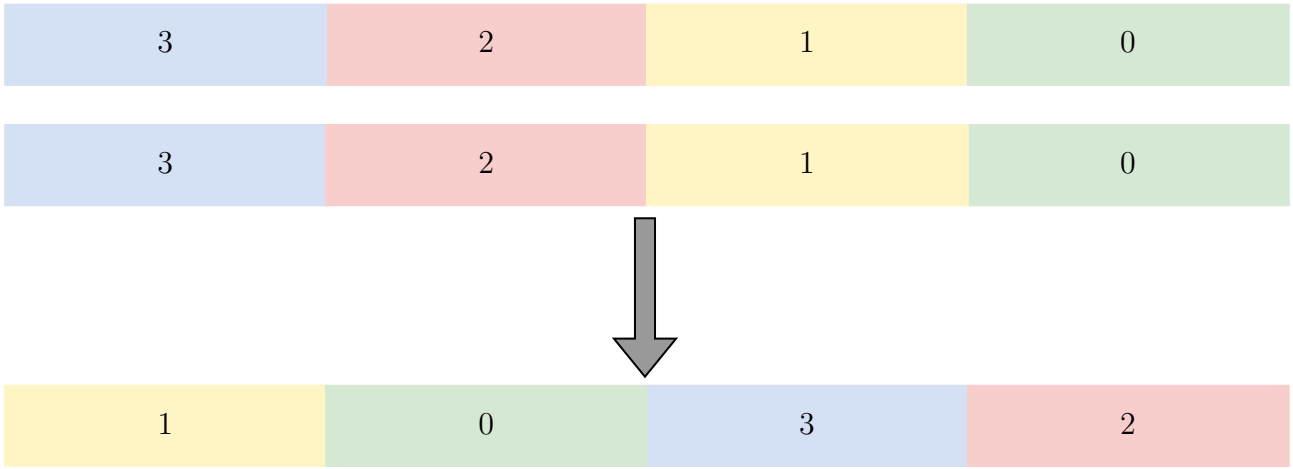

Figure 9: Step 1

2. We perform an addition with the vector from the previous step and the `acc` vector as shown in fig.10

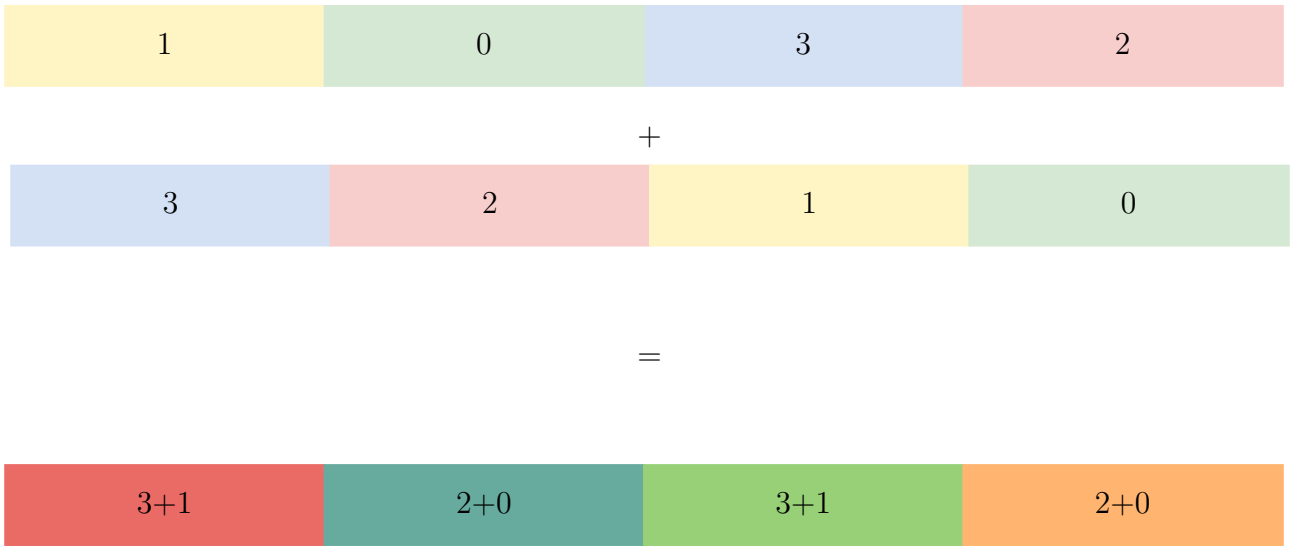

Figure 10: Step 2

3. The `_mm_movehdup_ps` function returns a vector that has the elements in odd indexed duplicated as shown in fig.11

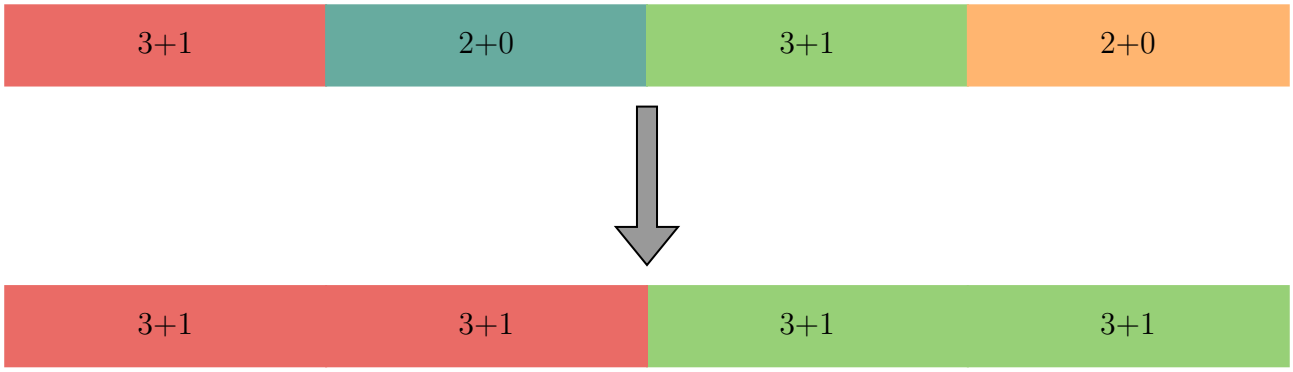

Figure 11: Step 3

4. Using the `_mm_add_ss` we add the elements in index 0 of the vectors created from the two previous step as shown in fig.12

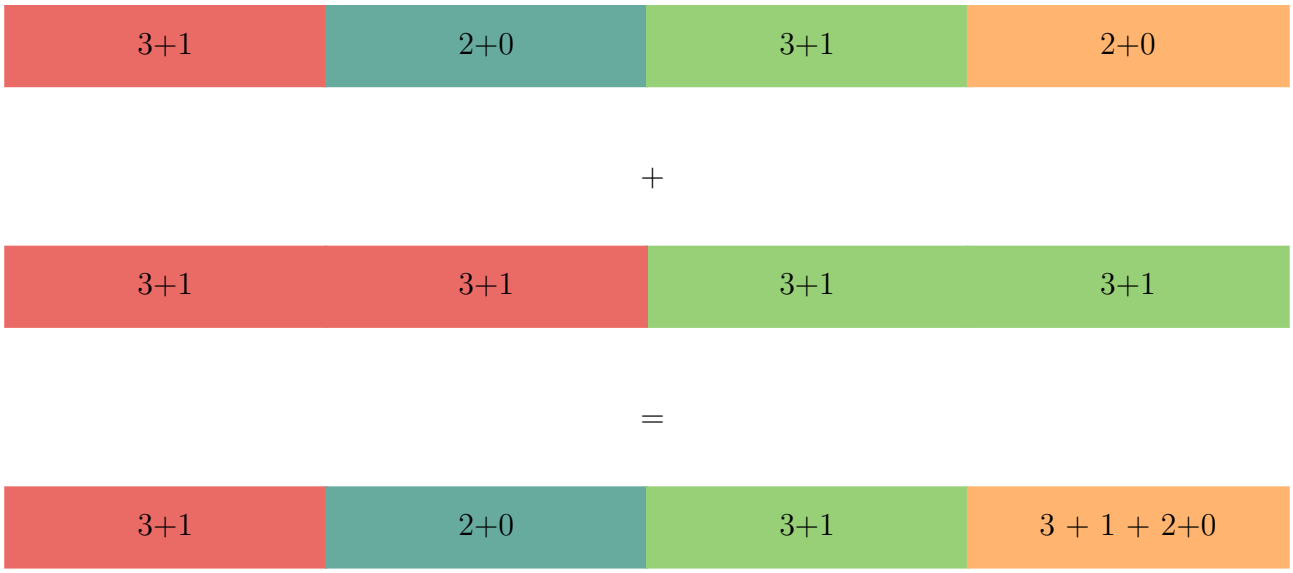

Figure 12: Step 4

The result is in index 0 of vector `r2`, which we extract using the `_mm_cvtss_f32` function.

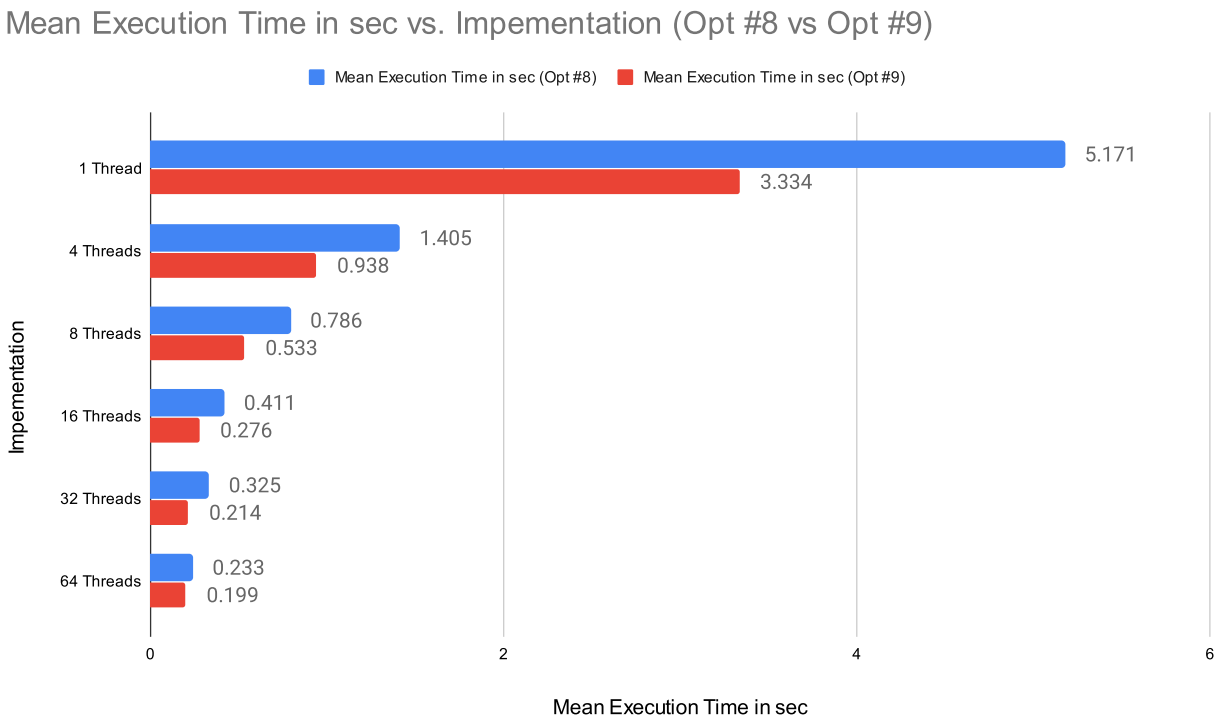

Figure 13: Mean Execution Time in sec for Optimizations #8 and #9

The use of SIMD units greatly benefited our code as seen in fig.13. The positive effect was most prominent in the cases where less threads are deployed, probably because there is a limited number of SIMD registers and the `euclidean_dist` function is simultaneously called by multiple threads that all need to use those registers.

5 Conclusions

Our findings demonstrated that increased parallelization does not always equate to increased speed. When we attempt to optimize small portions of the code, the execution time is often increased since the complexity associated with creating and destroying threads outweighs the benefits of parallelizing this little portion of the code. In order to prevent threads from being created and destroyed continuously, we should also try to parallelize larger sections of code and outer loops.

Fig.14 shows the mean execution time and standard deviations for all the described optimizations steps for the number of threads that correspond to the minimum execution time for this particular implementation (typically 64 threads unless noted otherwise).

Fig.15 shows the speedup gained for the different implementations and fig.16 zooms in on the mean execution times and standard deviations of only the successful optimizations (the ones that added an additional speedup over the previous optimization steps) to compare them.

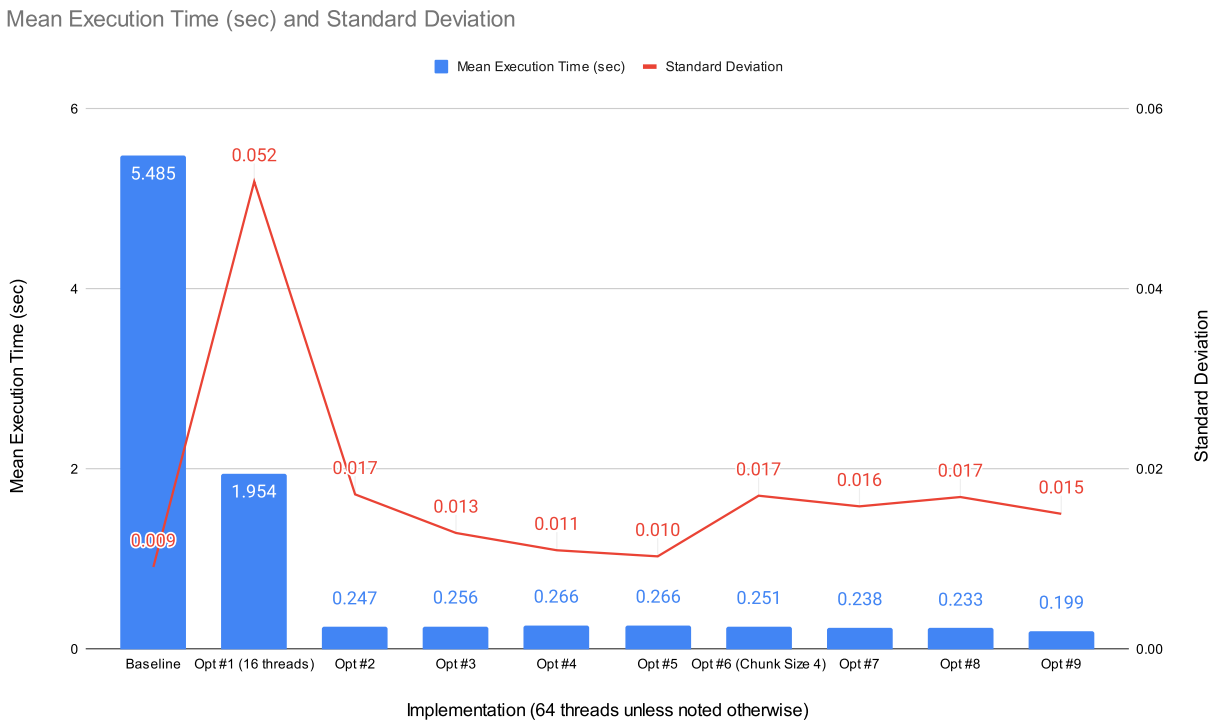

Figure 14: Mean Execution Time and Standard Deviations in sec for different implementations

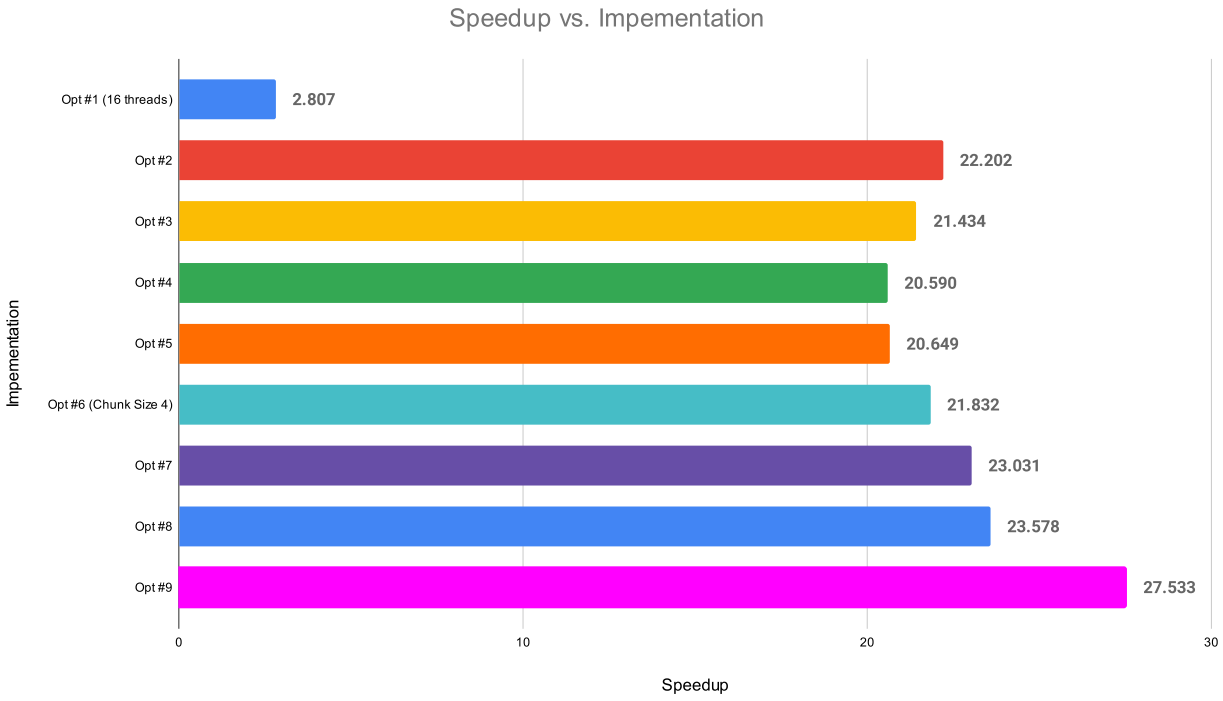

Figure 15: Speedup for different implementations (64 threads unless noted otherwise)

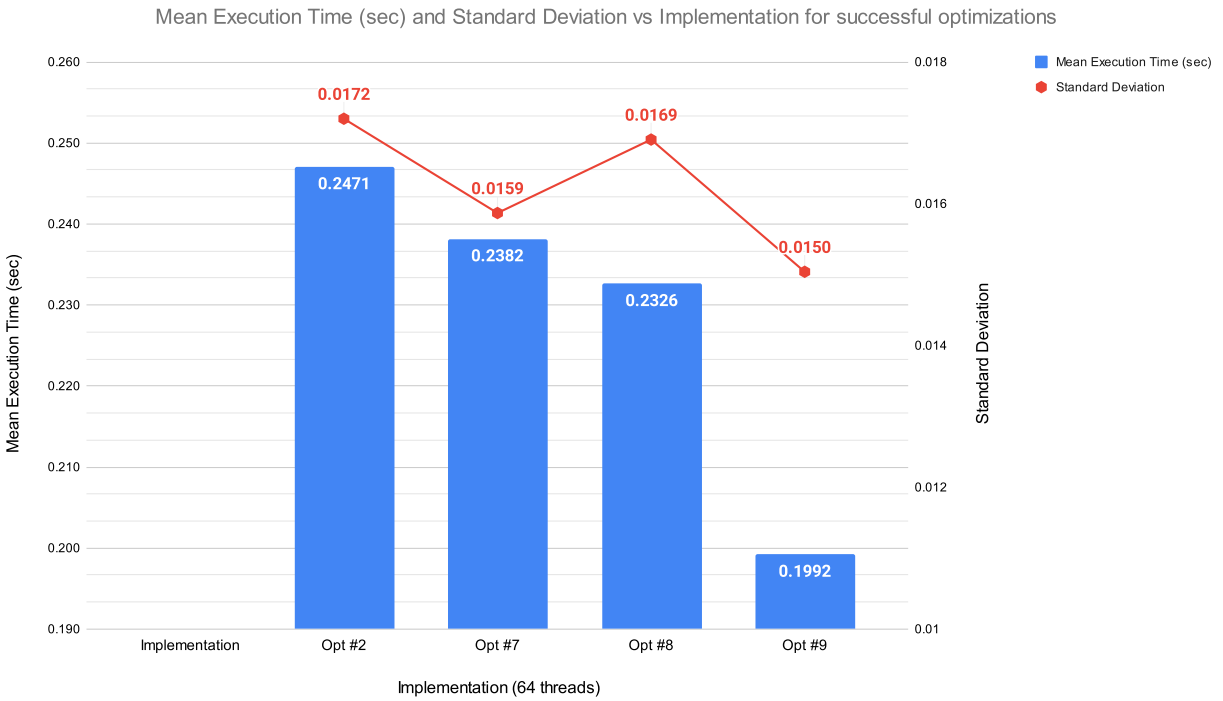

Figure 16: Mean Execution Time and Standard Deviations in sec for successful implementations