Electrical and Computer Engineering
University of Thessaly (UTH)

ECE415 - High Performance Computing (HPC)

Fall Semester — Educational year 2024-2025

Lab02

K-means algorithm parallelization using OpenMP

Dimitrios Tsalapatas - AEM: 03246

Georgios Kapakos - AEM: 03165

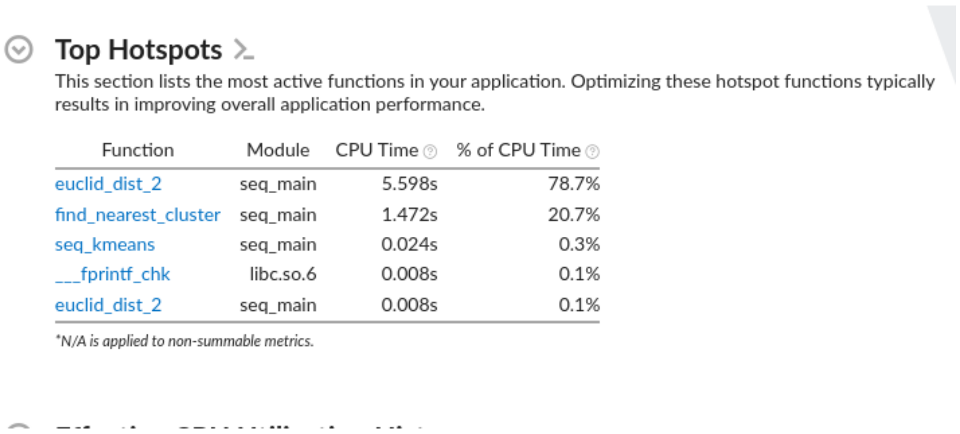
Contents

1 Introduction

K-means clustering aims to partition numObjs observations into numClusters clusters. Each of these objects is comprised of numCoords features, in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. K-means clustering minimizes within-cluster variances (squared Euclidean distances). We use an iterative technique to assign the objects to the nearest cluster, until the threshold is reached or the maximum counter of loop iterations is complete.

2 Profiling

For the profiling of the code we used the v-tune profiler. The profiler provides us with code parts which are the most time consuming. The parts that seem to be the most time-consuming are the for loops. We will try to optimize them using OpenMP pragmas.



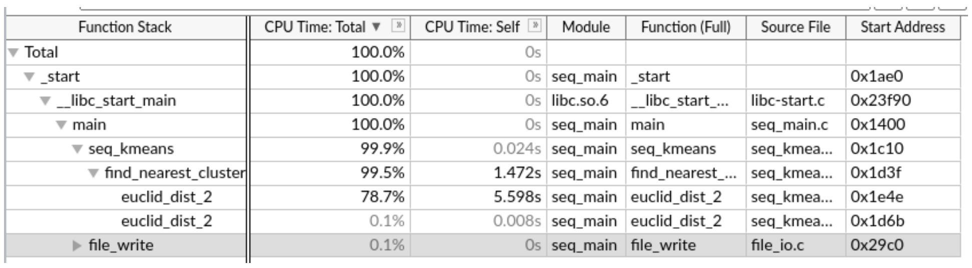
Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
euclid_dist_2	seq_main	5.598s	78.7%
find_nearest_cluster	seq_main	1.472s	20.7%
seq_kmeans	seq_main	0.024s	0.3%
___fprintf_chk	libc.so.6	0.008s	0.1%
euclid_dist_2	seq_main	0.008s	0.1%

*N/A is applied to non-summable metrics.

Figure 1: VTUNE: hotspots of original code



Function Stack	CPU Time: Total ▼	CPU Time: Self	Module	Function (Full)	Source File	Start Address
▼ Total	100.0%	0s				
▼ _start	100.0%	0s	seq_main	_start		0x1ae0
▼ __libc_start_main	100.0%	0s	libc.so.6	__libc_start_...	libc-start.c	0x23f90
▼ main	100.0%	0s	seq_main	main	seq_main.c	0x1400
▼ seq_kmeans	99.9%	0.024s	seq_main	seq_kmeans	seq_kmea...	0x1c10
▼ find_nearest_cluster	99.5%	1.472s	seq_main	find_nearest_...	seq_kmea...	0x1d3f
euclid_dist_2	78.7%	5.598s	seq_main	euclid_dist_2	seq_kmea...	0x1e4e
euclid_dist_2	0.1%	0.008s	seq_main	euclid_dist_2	seq_kmea...	0x1d6b
▶ file_write	0.1%	0s	seq_main	file_write	file_io.c	0x29c0

Figure 2: VTUNE: Top-down tree of runtime

3 Evaluation

For the evaluation of the code we run the program 12 times. We exclude the outliers of those epochs and then take the mean of the other 10. We run the program in the csl-venus environment for 1, 2, 4, 8, 14, 28, 56 threads (2 14-way multicore CPUs). The metrics we use for the evaluation of the program will be the average execution time and the standard deviation.

4 Makefile flags

We use the **icx** compiler with the optimization flags being:

$$OPTFLAGS = -qopenmp - fast - DNDEBUG - g$$

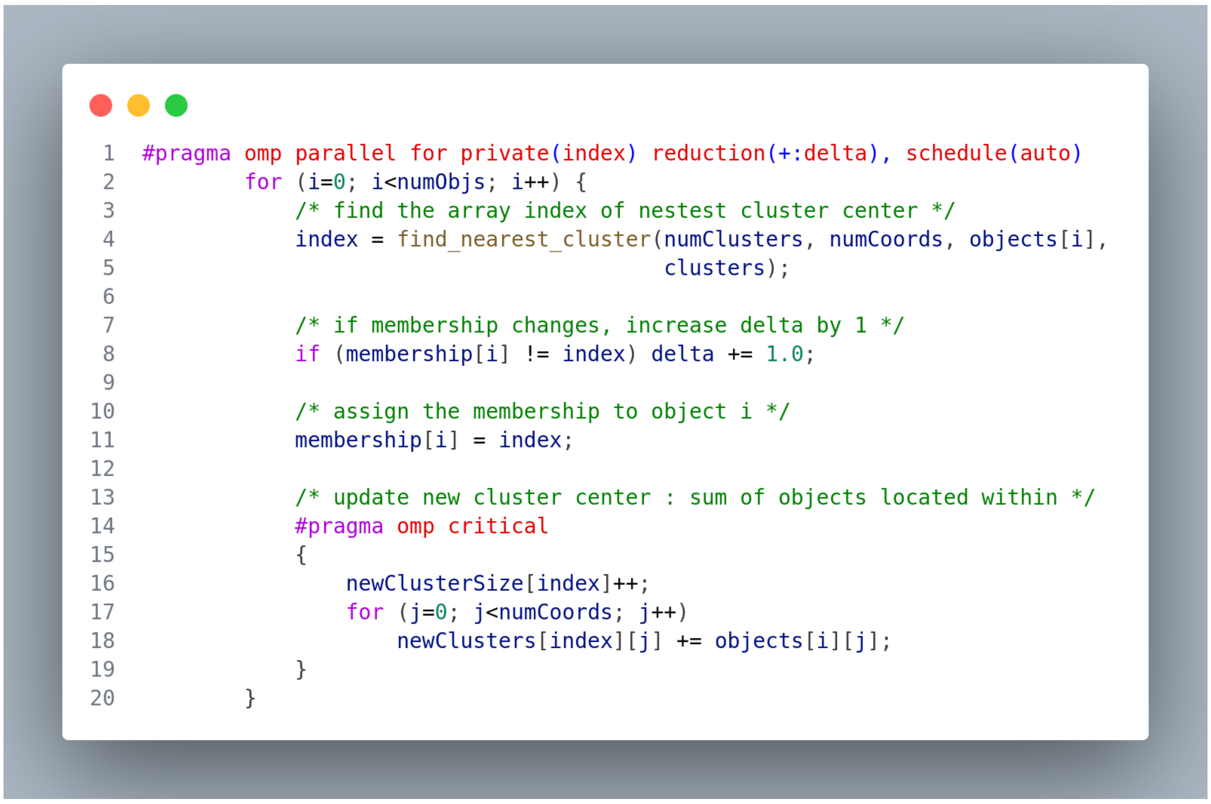
5 Optimization Steps

For the optimization steps we will commence our optimizations from the bigger loops in the largest function which calls the smaller functions inside. We decided to do after seeing from the V-tune profiler that the *euclidean_distance* function has the largest function. This function though gets called by two other functions that is why we chose to start optimizing from the function which calls all the other functions and later see if any optimizations could be made in the smaller functions.

5.1 1st Optimization

We observe that inside the function: seq_kmeans, the first for loop can be parallelized. We implement parallel for loop command. To avoid the race conditions inside the for loop we implement the following commands:

- **private index**: The index variable will be set as private because we need each thread to have its own independent copy of a variable, ensuring that the changes that will be made on this variable will not affect others. These private copies of the index variable exist only in the duration of the parallel for loop. Each thread can update index independently without affecting others.
- **reduction delta**: Because each thread will try to update the delta variable, the reduction clause will allow each thread to keep its own local copy of delta that is summed up at the end of the parallel region. This way, each thread independently increments its own delta without conflicting with other threads, and these values are combined once all threads finish executing.
- **schedule(auto)**: The schedule(auto) command gives the compiler the choice to select the best possible scheduling strategy for the program. The compiler decides this at runtime.
- **critical section**: A critical section occurs where a code block needs to be executed at one thread at a time. This is necessary to synchronize access to shared resources. The incrementation of the numClustersSize vector and the assignment of the newClusters array. The code there is executed sequentially and therefore the performance of the program does not improve.

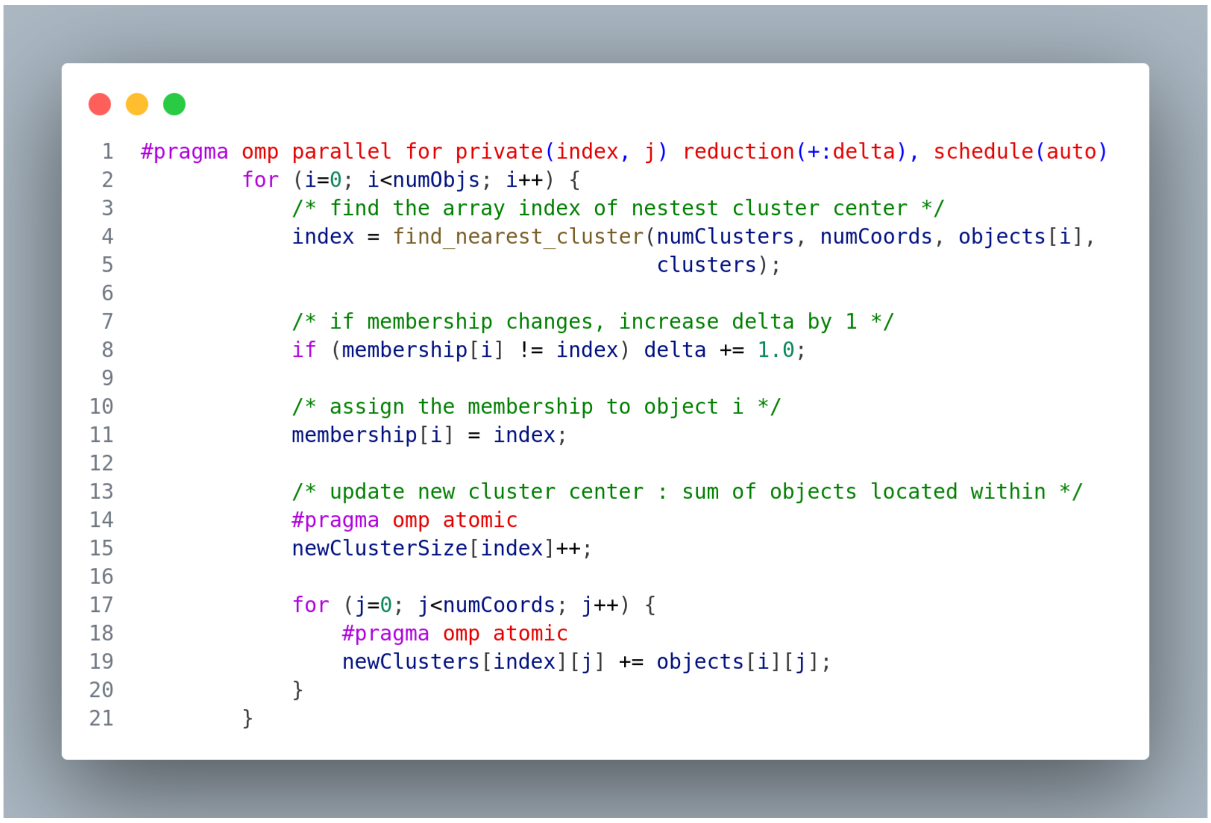


```c
#pragma omp parallel for private(index) reduction(+:delta), schedule(auto)
        for (i=0; i<numObjs; i++) {
            /* find the array index of nestest cluster center */
            index = find_nearest_cluster(numClusters, numCoords, objects[i],
                                         clusters);

            /* 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];
            }
        }
```

Figure 3: 1st Optimization code block

5.2 2nd Optimization

The critical pragma is used to protect a block of code, that involves multiple statements or variables. Where the atomic pragma is used for simple arithmetic operations. Atomic operations are implemented at the hardware level, that makes them faster than the critical block, which typically has a larger overhead. This concludes that the atomic pragma, if it is possible to be implemented to yield better execution time. We observe that the critical section in our code can be broken-down into two individual atomic sections. We first observed that the operations that were managed by critical where just simple arithmetic operations (additions) and the critical pragma could easily be replaced with atomic. Furthermore, the inner for loop because it is no longer inside a critical section must have the j variable as private to avoid any race conditions.



```
#pragma omp parallel for private(index, j) reduction(+:delta), schedule(auto)
        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
            newClusterSize[index]++;

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

Figure 4: 2nd Optimization code block

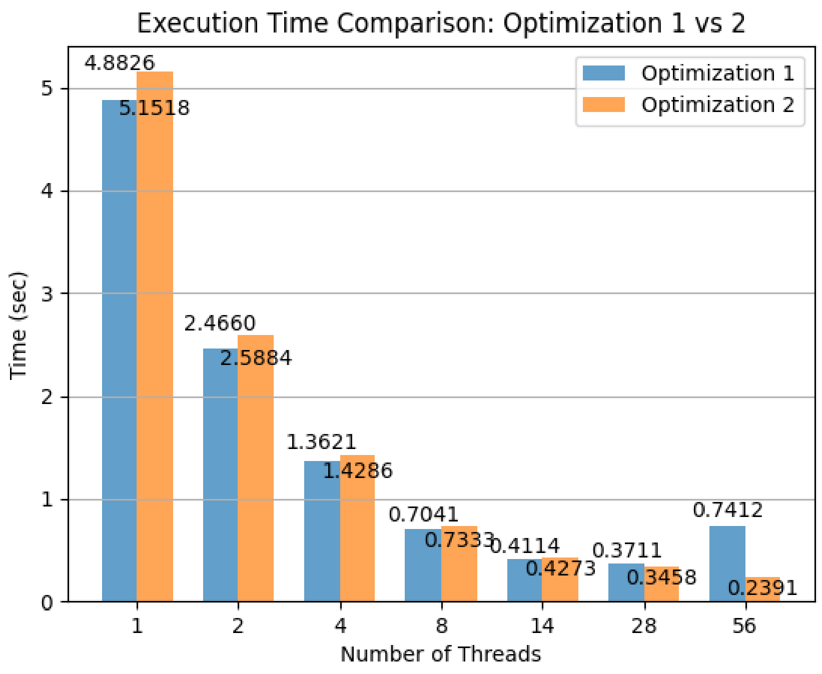
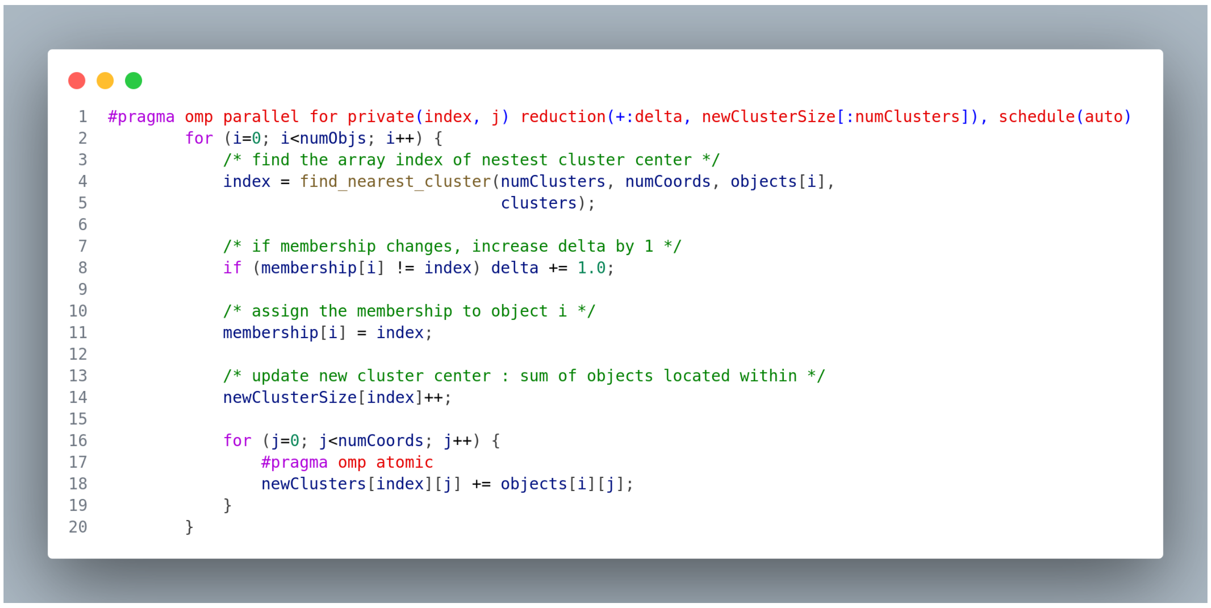


Figure 5: Comparison between optimisation #1 and #2

5.3 3rd Optimization

The third optimization step is to avoid use of 'critical' and 'atomic' in order to reduce the time that each thread should wait "block". We can use the operation reduction on the 1-D vector *newClusterSize[numClusters]* because each thread just adds 1 on this position of array. Using reduction each thread will create a local copy of this position of array in order to increase it, and in the end it will add all local variables to *newClusterSize*.



```
#pragma omp parallel for private(index, j) reduction(+:delta, newClusterSize[:numClusters]), schedule(auto)
        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];
            }
        }
```

Figure 6: 3rd Optimization code block

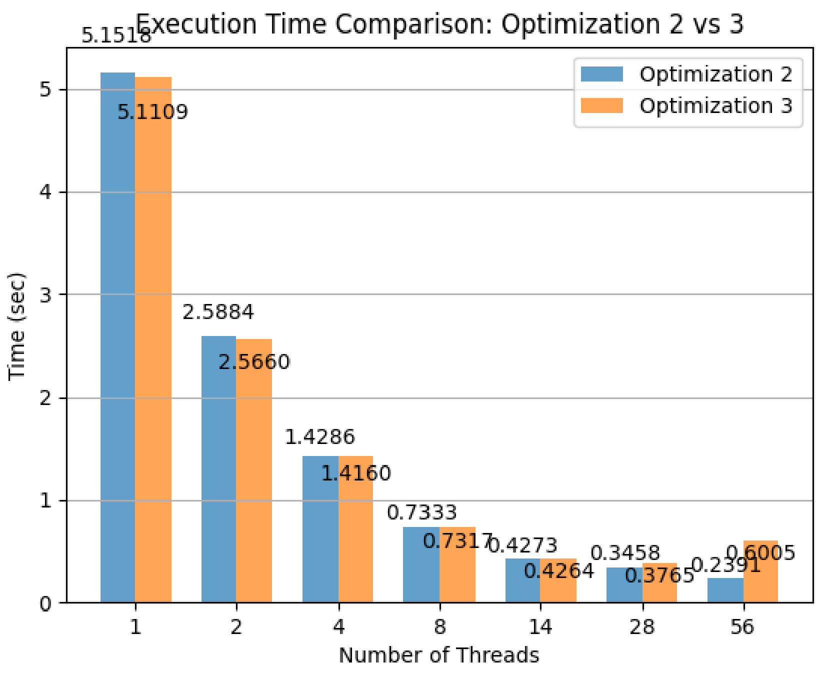
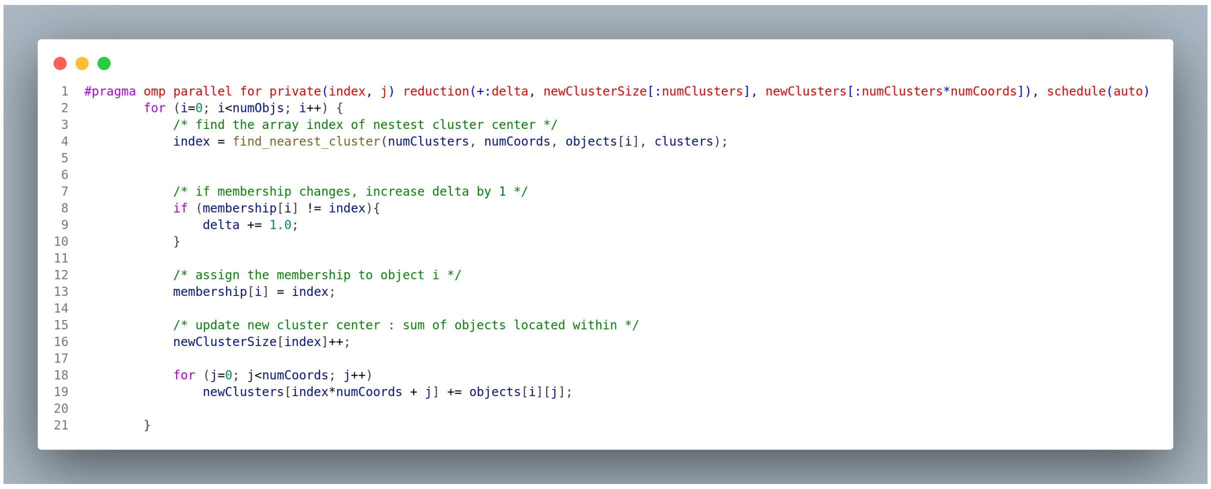


Figure 7: Comparison between optimisation #2 and #3

5.4 4th Optimization

The 2-D array *newClusters[numClusters][numCoords]*, is initially vectorized and later reduced with the reduction pragma. Only vectors can be reduced not 2 dimensional arrays. That is the main case for the transformation from a 2-D to an 1-D array. We make some alterations based on the access patterns on the now 1-D: *newClusters[numClusters * numCoords]*, inside the inner for loop. Further alterations, have to be made outside the do-while loop which contains the for loops to accommodate for this modification which are memory allocation and array initialization.



```
#pragma omp parallel for private(index, j) reduction(+:delta, newClusterSize[:numClusters], newClusters[:numClusters*numCoords]), schedule(auto)
        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];

        }
```

Figure 8: 4th Optimization code block

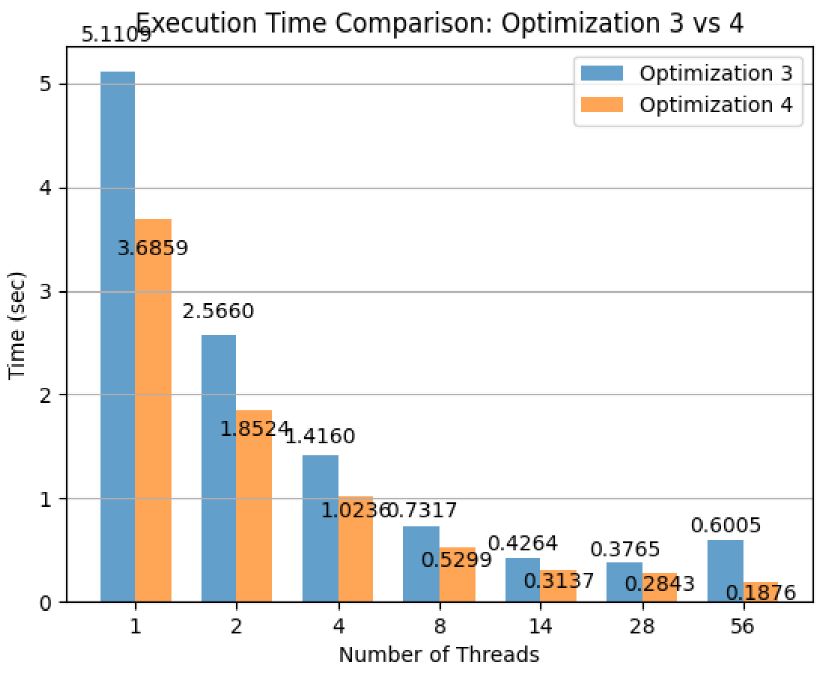
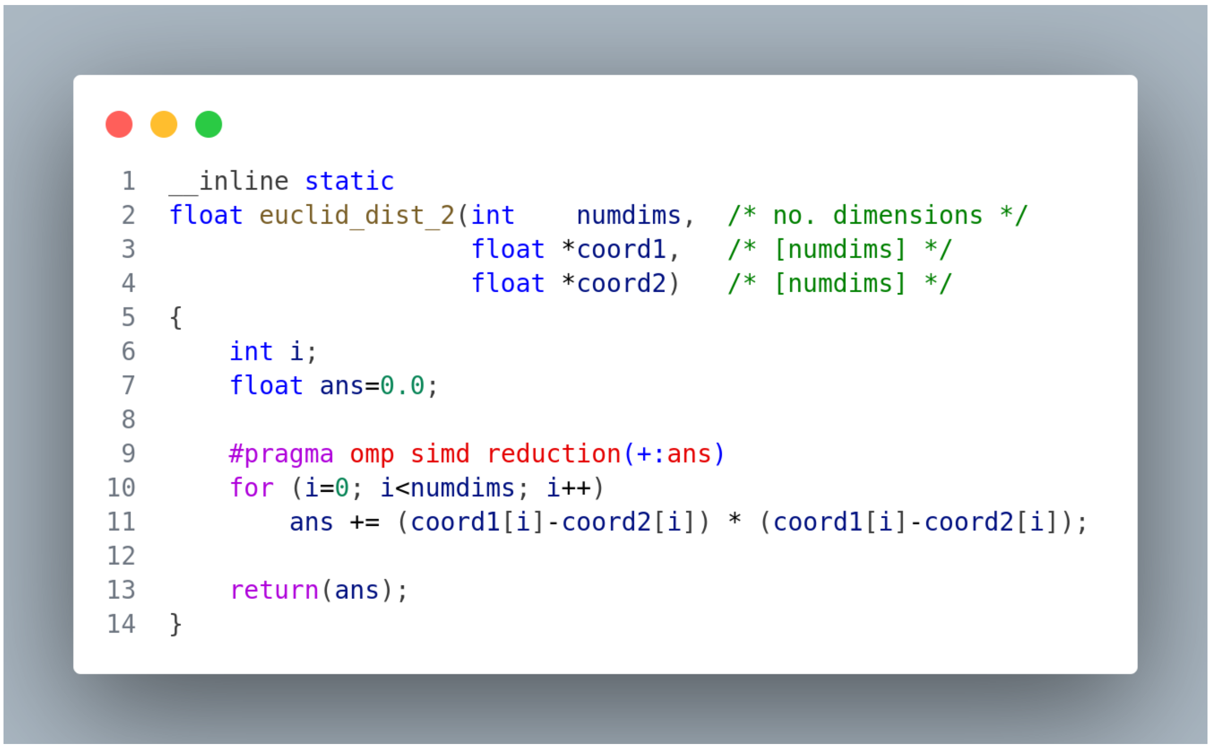


Figure 9: Comparison between optimisation #3 and #4

5.5 5th Optimization

The only loop left for optimization is the for loop inside the *euclid_dist_2* function. This function cannot be parallelized, so use the pragma simd, to try to optimize it sequentially as good as we can. The compiler attempts to transform the loop into a vectorized version, where each iteration of the loop can be executed in parallel by the SIMD hardware in the CPU, it is especially useful in this loop as only one operation is done per iteration.



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

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

    return(ans);
}
```

Figure 10: 6th Optimization code block

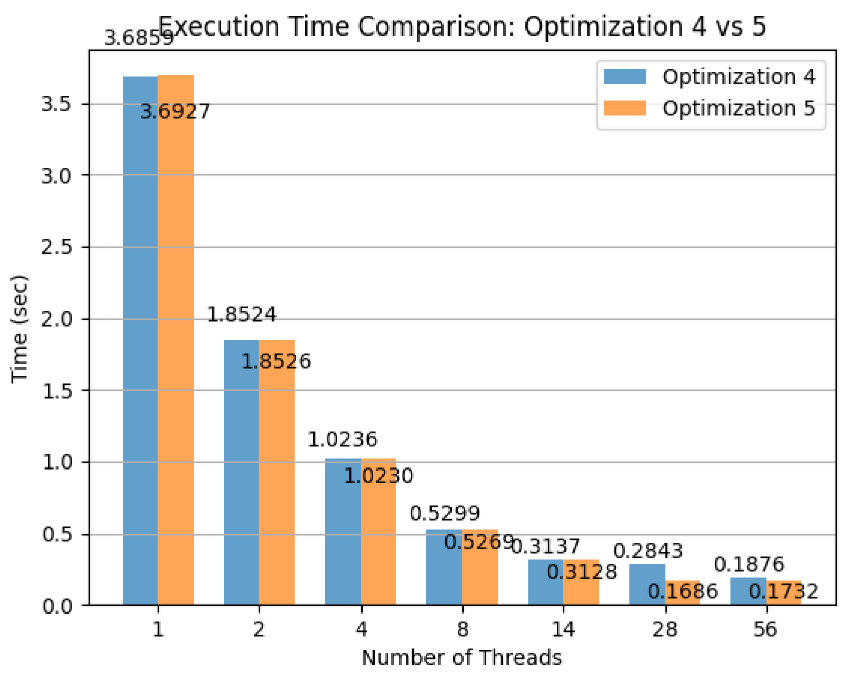


Figure 11: Comparison between optimisation #4 and #5

5.6 6th Optimization

In the last step of our optimizations we try some environmental variables. The ones we try are:

- ***OMP_WAIT_POLICY***: The *OMP_WAIT_POLICY* specifies how threads should behave when they are idle, particularly after reaching synchronization points like barriers or waiting for work in parallel regions. This variable affects how threads wait for new tasks, potentially influencing CPU performance. The two choices for this variable are ACTIVE, which means that threads remain in an active waiting state and PASSIVE, which means that threads go into a sleep state (or ”passive wait”) when idle, freeing up CPU resources for other tasks.

- ***OMP_PROCESSOR_BIND***: *OMP_PROC_BIND* controls thread binding — specifically, how threads are bound to CPU cores. Binding threads to specific cores can improve performance by reducing cache misses and optimizing data locality, as threads consistently use the same CPU cores throughout execution.

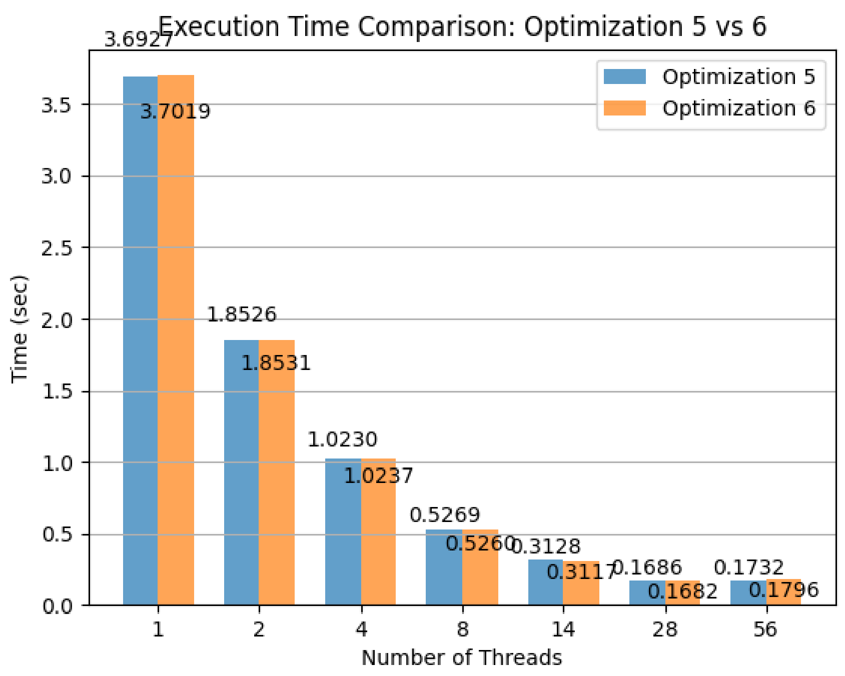


Figure 12: Comparison between optimisation #5 and #6

5.7 Further Optimizations

After realizing that we can no longer optimize the for loops inside our *seq_kmeans* function we proceed with the *find_nearest_cluster* function. We have already created a struct which contains both the minimum distance and the minimum index inside. A struct may result in faster access times, but this doesn't make a difference in the speed of the code. Although, if we parallelize the for loop in the *find_nearest_cluster* function and avoid the race conditions by using the reduction pragma with the min operation, this will yield a reduced time. The reduction operation could not work on the certain code block because the if statement controls two variables. This troubled us and we could not proceed with the optimization of the for loop.

6 Conclusion

Finally, we observed that after the 3rd Optimization the other optimizations did not make any significant difference in the decrement of the time. Parallelization of the code made sense when parallelizing large and expensive portions, where parallelizing small blocks of code did not yield any optimizations, sometimes yielding worse times.

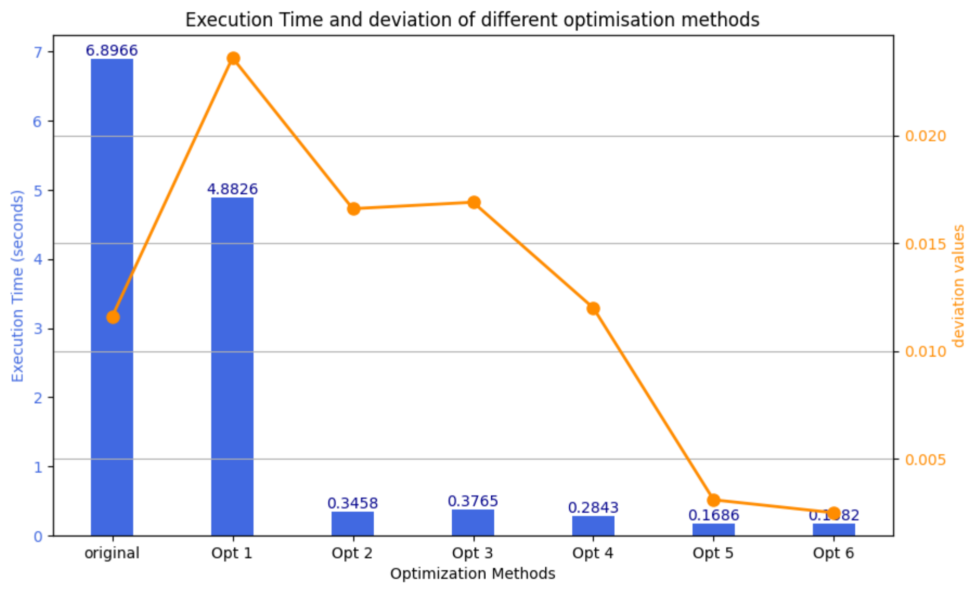


Figure 13: Impact of optimizations on runtime

Optimisation Steps	**Execution Time (Venus)**													
Number of Threads	1		2		4		8		14		28		56	
	Time	Diviation	Time	Diviation	Time	Diviation	Time	Diviation	Time	Diviation	Time	Diviation	Time	Diviation
Original	6,8966	0,0116	-	-	-	-	-	-	-	-	-	-	-	-
Parallel first for, private index, reduction delta, critical inside for #1	4,8826	0,0135	2,466	0,0068	1,3621	0,0022	0,7041	0,0011	0,4114	0,002	0,3711	0,0236	0,7412	0,2582
Change newCluserSize and inner loop from critical section to atomic #2	5,1518	0,039	2,5884	0,0165	1,4286	0,0045	0,7333	0,0038	0,4273	0,0015	0,3458	0,0166	0,2391	0,024
remove newClusterSize from critical and add it as reduction #3	5,1109	0,0071	2,566	0,0051	1,416	0,004	0,7317	0,03	0,4264	0,0017	0,3765	0,0169	0,6005	0,2501
Reduction for 2D newClusters and add it as reduction #4	3,6859	0,0275	1,8524	0,0054	1,0236	0,0037	0,5299	0,002	0,3137	0,0034	0,2843	0,012	0,1876	0,0223
OMP_WAIT_POLICY = active #5	3,6927	0,0191	1,8526	0,0116	1,023	0,0025	0,5269	0,001	0,3128	0,0025	0,1686	0,0031	0,1732	0,0126
OMP_PROCESSOR_BIND #6	3,70	0,0208	1,8531	0,0057	1,0237	0,0042	0,526	0,0019	0,3117	0,0012	0,1682	0,0025	0,1796	0,0215

Table 1: Table comparing all the optimisation steps on different number of threads