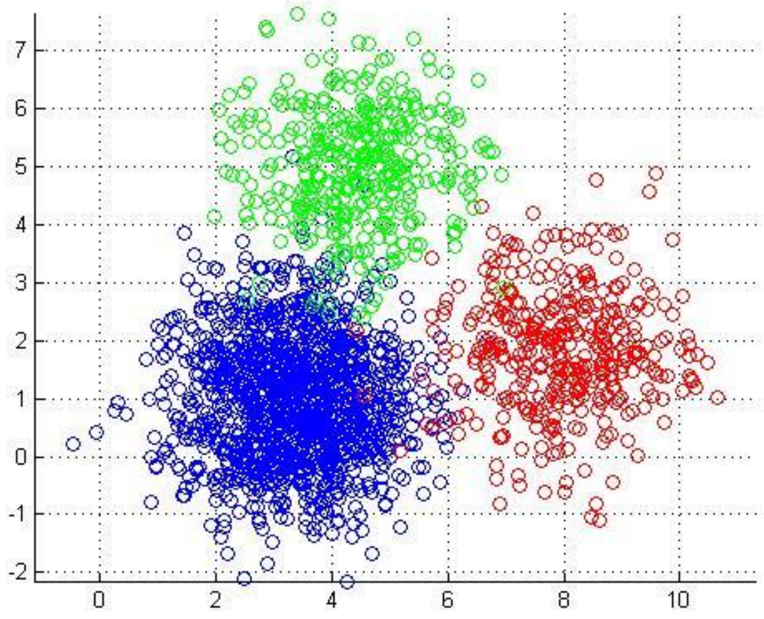
ECE415 High Performance Computing

Lab Assignment 2

Parallelization of a serial K-means clustering application using OpenMP

Kyritsis Spyridon 2697

Karaiskos Charalampos 2765



Project Description

The goal of this project is to optimize the provided K-means implementation application by parallelizing it using OpenMP.

K-means clustering is a method of vector quantization, originally from signal processing, that aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster. This results in a partitioning of the data space into Voronoi cells. k-means clustering minimizes within-cluster variances (squared Euclidean distances).

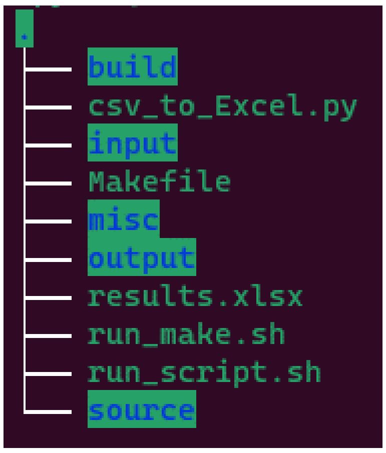
As per the assignment's requirements all our tests were performed using the Intel compiler icc on the csl-artemis system.

Deliverable Directory Structure

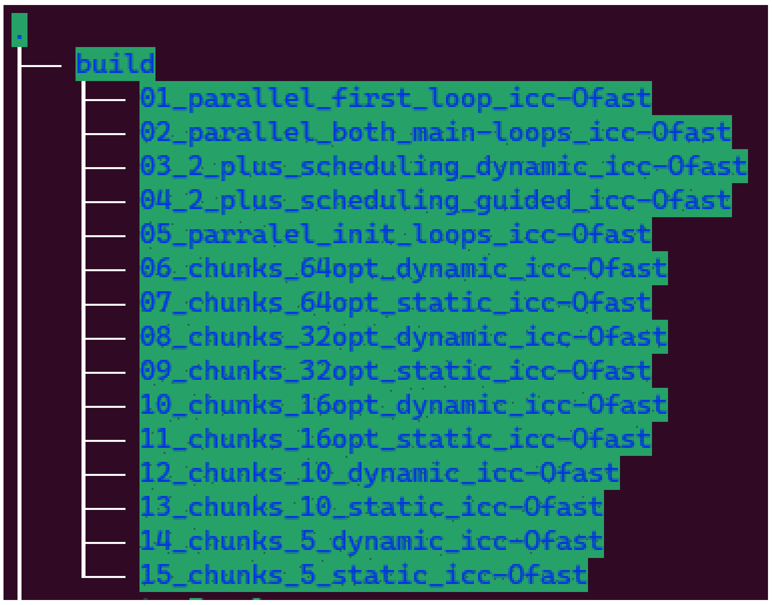
To make sure our deliverable is as clean as possible we organized our directory structure as follows:

- input/
    - The input directory contains all the provided input files for the tests to run. Also the .membership and .cluster_centers files that are the results of the last ran executable are stored there.
- source/
    - The source directory contains subdirectories each containing the k-means project with a different set of parallelizations performed on the seq_kmeans.c file. Each subdirectory name begins with the number signifying the order in which the parallelizations were performed. The rest of the subdirectory name is a brief description of the parallelization performed.
- build/
    - The build subdirectory contains all binary files produced during the make process. The executable files names are the name of the build directory their C source code originated from, with the name of the compiler and the optimization flag used appended in the end.

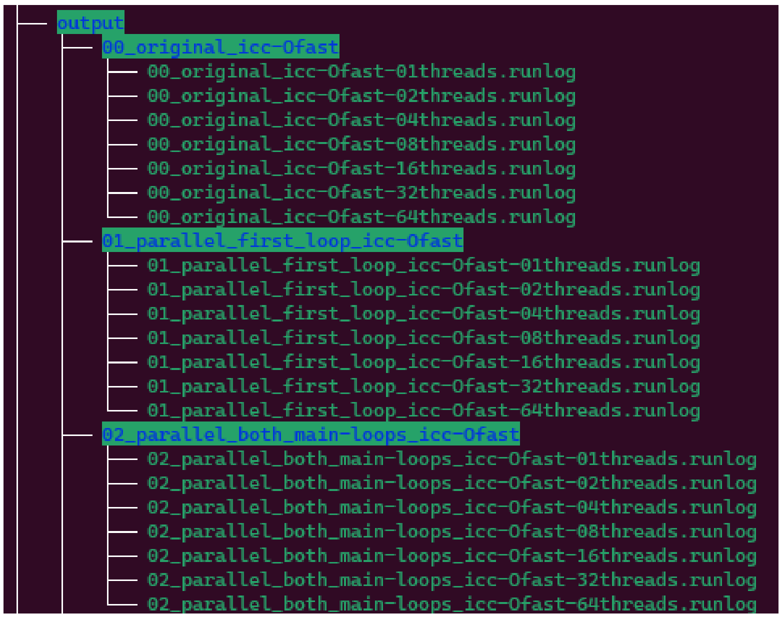
- output/
    - The output subdirectory contains a subdirectory for each executable file present in build/. Each subdirectory contains a .runlog file for every number of threads tested, where the computation time for each execution is stored.
- The top-level directory contains a suite of scripts we will be discussing later as well as the Makefile for the project, a .membership and a .cluster_centers file that came from the original project that were used to test the validity of our outputs.



```
.
├── build
├── csv_to_Excel.py
├── input
├── Makefile
├── misc
├── output
├── results.xlsx
├── run_make.sh
├── run_script.sh
└── source
```



```
.
├── build
│   ├── 01_parallel_first_loop_icc-Ofast
│   ├── 02_parallel_both_main-loops_icc-Ofast
│   ├── 03_2_plus_scheduling_dynamic_icc-Ofast
│   ├── 04_2_plus_scheduling_guided_icc-Ofast
│   ├── 05_parralel_init_loops_icc-Ofast
│   ├── 06_chunks_64opt_dynamic_icc-Ofast
│   ├── 07_chunks_64opt_static_icc-Ofast
│   ├── 08_chunks_32opt_dynamic_icc-Ofast
│   ├── 09_chunks_32opt_static_icc-Ofast
│   ├── 10_chunks_16opt_dynamic_icc-Ofast
│   ├── 11_chunks_16opt_static_icc-Ofast
│   ├── 12_chunks_10_dynamic_icc-Ofast
│   ├── 13_chunks_10_static_icc-Ofast
│   ├── 14_chunks_5_dynamic_icc-Ofast
│   └── 15_chunks_5_static_icc-Ofast
```



```
├── output
│   ├── 00_original_icc-Ofast
│   │   ├── 00_original_icc-Ofast-01threads.runlog
│   │   ├── 00_original_icc-Ofast-02threads.runlog
│   │   ├── 00_original_icc-Ofast-04threads.runlog
│   │   ├── 00_original_icc-Ofast-08threads.runlog
│   │   ├── 00_original_icc-Ofast-16threads.runlog
│   │   ├── 00_original_icc-Ofast-32threads.runlog
│   │   └── 00_original_icc-Ofast-64threads.runlog
│   ├── 01_parallel_first_loop_icc-Ofast
│   │   ├── 01_parallel_first_loop_icc-Ofast-01threads.runlog
│   │   ├── 01_parallel_first_loop_icc-Ofast-02threads.runlog
│   │   ├── 01_parallel_first_loop_icc-Ofast-04threads.runlog
│   │   ├── 01_parallel_first_loop_icc-Ofast-08threads.runlog
│   │   ├── 01_parallel_first_loop_icc-Ofast-16threads.runlog
│   │   ├── 01_parallel_first_loop_icc-Ofast-32threads.runlog
│   │   └── 01_parallel_first_loop_icc-Ofast-64threads.runlog
│   ├── 02_parallel_both_main-loops_icc-Ofast
│   │   ├── 02_parallel_both_main-loops_icc-Ofast-01threads.runlog
│   │   ├── 02_parallel_both_main-loops_icc-Ofast-02threads.runlog
│   │   ├── 02_parallel_both_main-loops_icc-Ofast-04threads.runlog
│   │   ├── 02_parallel_both_main-loops_icc-Ofast-08threads.runlog
│   │   ├── 02_parallel_both_main-loops_icc-Ofast-16threads.runlog
│   │   ├── 02_parallel_both_main-loops_icc-Ofast-32threads.runlog
│   │   └── 02_parallel_both_main-loops_icc-Ofast-64threads.runlog
```

Experimental Methodology

To ensure our results reveal the actual performance of our application we performed each test a total of 22 times. We then proceeded to remove the best and worst result we got to make sure random events such as page faults and interrupts, due to operating system processes do not skew our measurements, leaving us with 20 runs of each experiment.

We also made sure that the csl-artemis system was as unoccupied as possible at the time the experiments were performed.

In order to streamline our experimental methodology, a suite of scripts was created in both Bash and Python.

The given Makefile was expanded to include extra functionality and functions as a master script. This means that we can call all other created scripts through the make <command> syntax.

The commands supported by the Makefile are:

- make clean
    - Removes all created executables and output files (images and logs)
- make all
    - Calls the run_make.sh Bash script. This script takes all source C files and creates executables. The script is configurable to compile the C files with an arbitrary number of compilers and optimization flags, as long as they have the appropriate flags set.
- make run
    - Calls the run_script.sh Bash script. This script runs 22 experiments for each thread number that was tested for each executable in build/, created by the make all command. The script is configurable to run any number of experiments specified by the user. The script is configured to create all necessary directories under output/ and log files from the experiments. The output of the executables is redirected to a log file, where it is stored in a CSV format for easier data processing.
- make excel
    - Calls the csv_to_Excel.py Python script. This script takes as input the input, build, and source directories as well as an optional parameter with number of test runs, if a value different than 22 was specified during the make run command. The script creates an Excel

spreadsheet with all the results sorted across compilers and optimization levels. The Python script reads all CSV log files finds and drops the best and worst rows, so as not to skew our results. Furthermore, the script writes the Excel formulas to calculate the average execution time of each test and the standard deviation of the results.

- **Note:** For the Python script to work we needed to install the following on our Ubuntu machine
  - `apt-get install python3.9`
  - `pip install openpyxl`
  - `pip install pandas`
- **Note:** We encountered a problem with the openpyxl framework where the formula for the standard deviation did not work in the actual Excel spreadsheet. This happened due to the openpyxl framework internally using a deprecated version of the STDDEV.P formula which included an extra character. This had to be removed by hand in the Excel spreadsheets via find and replace.

Profiling

Prior to running our optimizations, we first made sure to profile the given code, so that we identify the major bottlenecks early and focus on that. The profiling took place on the csl-artemis server which has the following specifications:

- 2 Intel Xeon E5-2683 v4 CPUs clocked at 2.10GHz
  - Each CPU has 16 cores and both CPUs are 2-way hyperthreaded, thus totaling 64 logical cores
- 128 GBs of RAM
- Ubuntu 20.04.5 LTS (GNU/Linux 5.4.0-126-generic x86_64)

Profiling revealed quite a few things about the given application.

Firstly, the code is highly optimized. Memory accesses and initializations are coded in such a way that they are as optimal as possible.

And secondly, the most major observation was that 95% percent of the time was spent in function Euclid_dist2(). This meant that all our effort would focus on optimizing this specific part of the given code as much as possible.

Parallelization

To perform our parallelizations, we used the given (serial) code for the K-means algorithm as the control version, and optimized the code based on that. Below we can find a list with all performed optimizations. The optimizations were applied in the order they are presented and in an additive manner, meaning that each test builds on top of each predecessor.

In this section we will describe what those parallelizations are, and in the next section we will discuss our statistical observations.

- **test1:** For test 1 we initially attempted to parallelize the loop inside the euclid_dist2 function, since the profiling revealed that the runtime spent about 98% of the time in that function. The results though were substantially worse than before we parallelized it. We came into the conclusion that this was due to the large overhead created by the number of times the worker threads had to be created. What we ended up doing for the first test, was parallelizing the first for loop in the do while loop which contains all the calls to the find_nearest_cluster() function, which itself contains the calls to function euclid_dist2().

- **test2:** For test 2 we parallelized the other for loop in the do while and placed it into the same parallel region as the other for loop thus parallelizing both loops while only creating the worker threads once.

- **test3, test4:** For tests 3 and 4 we tested using a different scheduling method for the parallelization than the default static method, specifically dynamic in 3 and guided in 4. Dynamic seemed to provide the better results, though only slightly, so we went forward keeping that.

- **test5:** For test 5 we parallelized the loops that initialize the arrays in the start of the algorithm. We did that by dividing them in sections, so that both initializations would run at the same time, thus saving even more time than simply parallelizing them one by one.

After that, we attempted to parallelize the whole do while loop, but came into the conclusion that although this was possible, the loop variable would be a critical section and thus a bottleneck for our application. This happens due to the application being instructed to exit after a specified number of iterations when the clustering results do not converge. Since this was

The rest of the tests we performed focus on how different chunk sizes affect the quality of our results. We organized our tests so that starting from **test6** and onwards every even numbered test refers to dynamic scheduling and every odd numbered test refers to static scheduling.

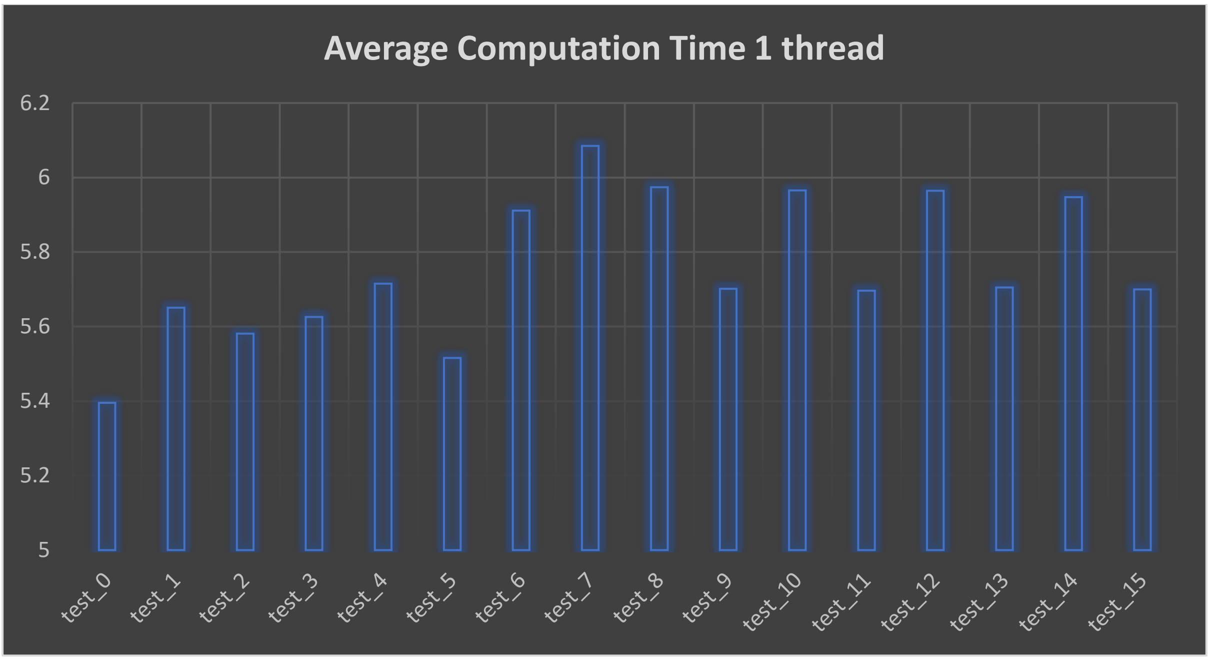
- **tests6-11:** For those tests we divided the workload into chunks so that each thread gets as even of a workload as possible. This was done mostly targeting the higher thread runs and thus only performed for 64, 32 and 16 threads respectively.
- **tests12-15:** For those tests we tried to divide the workload into arbitrarily smaller chunks (specifically 5 and 10) in the hopes that this will produce better coordination among threads and thus better results.

Graphical Result Representation

Note: For all the graphs presented below except the first one we exclude test_0. Test_0 is the given serial implementation which only exists as a benchmark. Since it heavily skews the graphs results, we decided to omit it from the graphs to better observe and understand our results.

The result section presents graphs comparing the average execution times of the different implementations through a series of different thread numbers.

- **Single-threaded execution:** Single threaded execution presents the greatest variance across the results we acquired. The clear winner here is the given serial benchmark implementation since it does not spawn any threads and thus has no overhead. All variation for this test can be attributed to the overhead by the OpenMP function calls and synchronization.



Below we present the results for the actually multithreaded implementations. It is noteworthy that with very minor exceptions the trends in our results are remarkably stable.

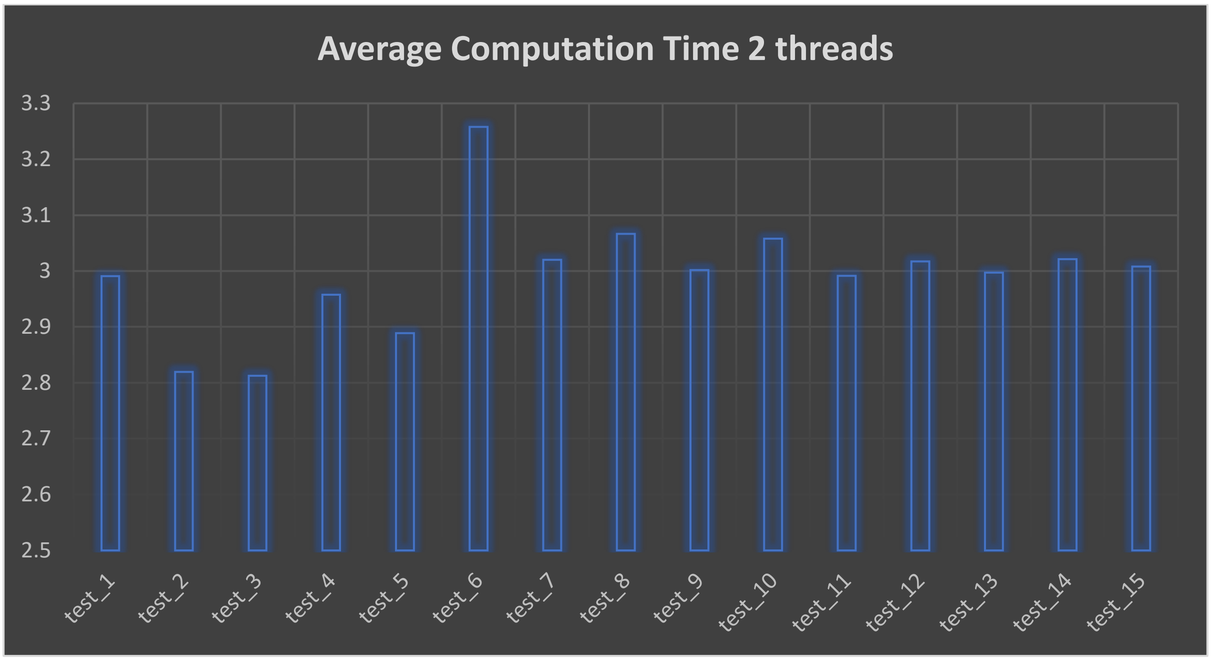
The trends we observed can be summarized as follows:

- The variation between the best and worst tests is on average around 0.15 seconds.
- Dynamic scheduling produces the best results albeit with a very small margin versus static scheduling.
- The default chunk sizes seem to be working better than any attempted optimization.
- That being said, **test3** and **test5** consistently produce the best results across all numbers of tests.
- A lot of the variation can be attributed to random events in the Operating System, which can be clearly viewed through the heatmap in the table of the results, that is included in the Excel spreadsheet in the deliverable folder.

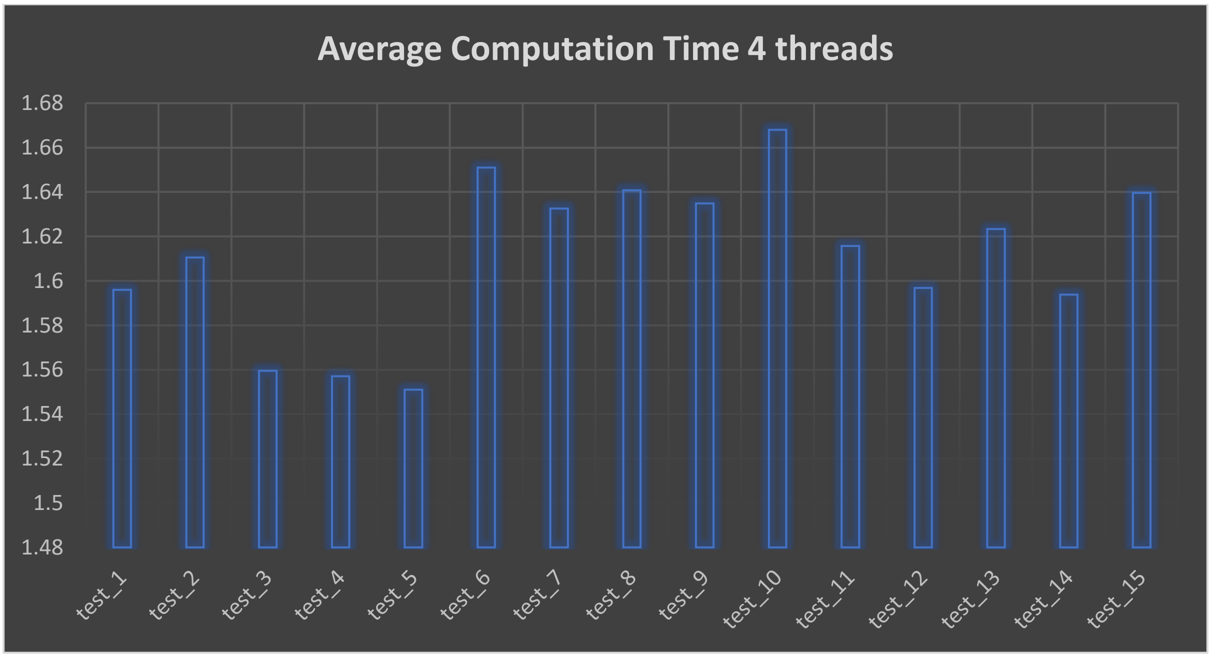
However, there is one exception in the above observations. This occurs in the section of the experiments where we use 32 and 64 threads respectively. Specifically, this refers mostly to **test10** and **test11** and to a lesser degree to **test8** and **test9**. In these cases, we observe the biggest jumps in variation with it being around 0.9 seconds on average! We can deduce that the chunk size that aimed to improve the 16-threaded execution leads to a very unfortunate miscoordination between the threads, which leads to a massive increase in execution time.

Below we present our results for 2, 4, 8, 16, 32 and 64 threads respectively, as well as an "aerial" view of the heatmap discussed above (for the 64-threaded tests only due to the space constraints).

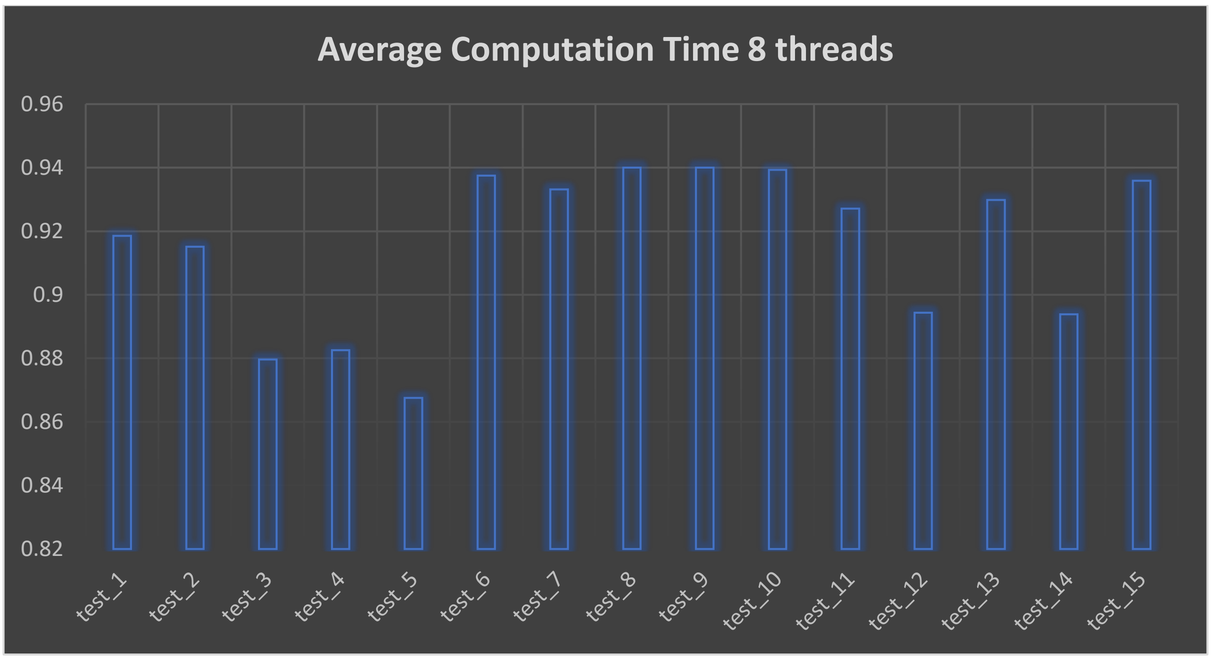
- **2-threaded execution:**



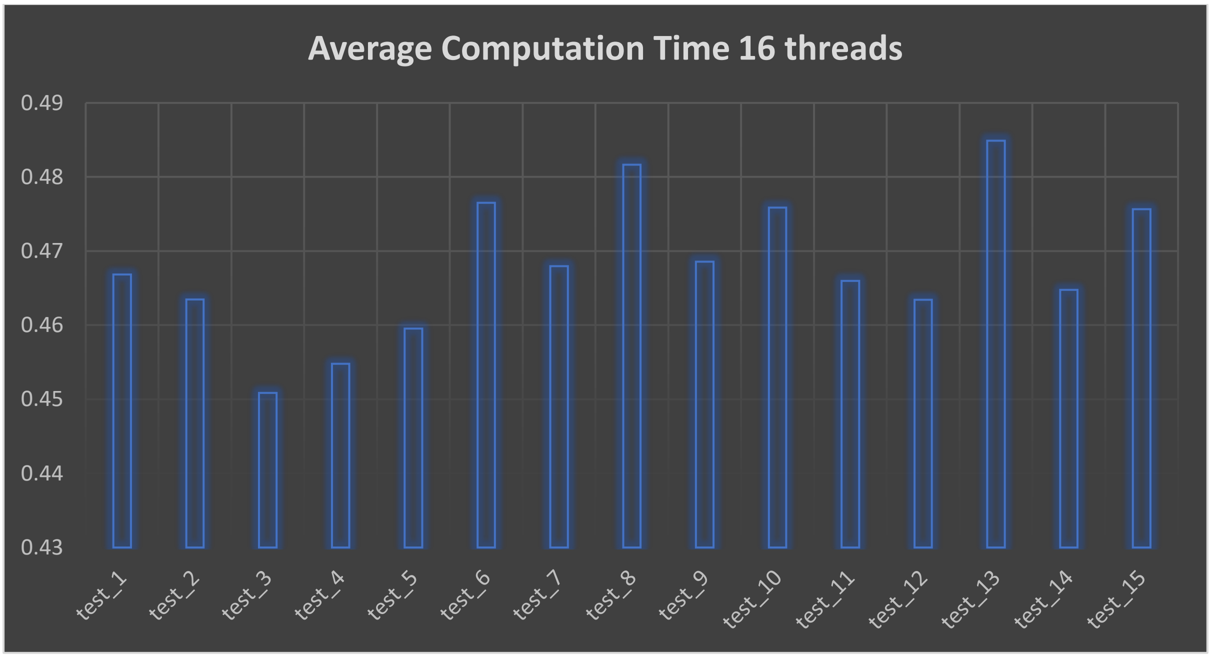
- **4-threaded execution:**



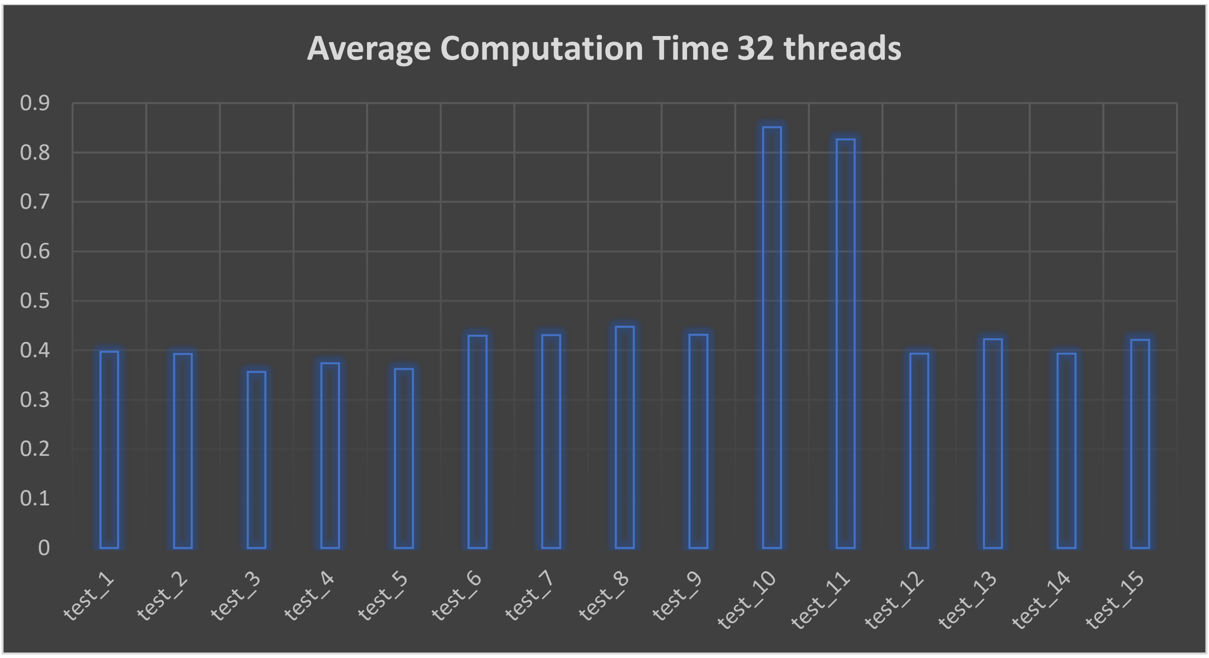
- **8-threaded execution:**



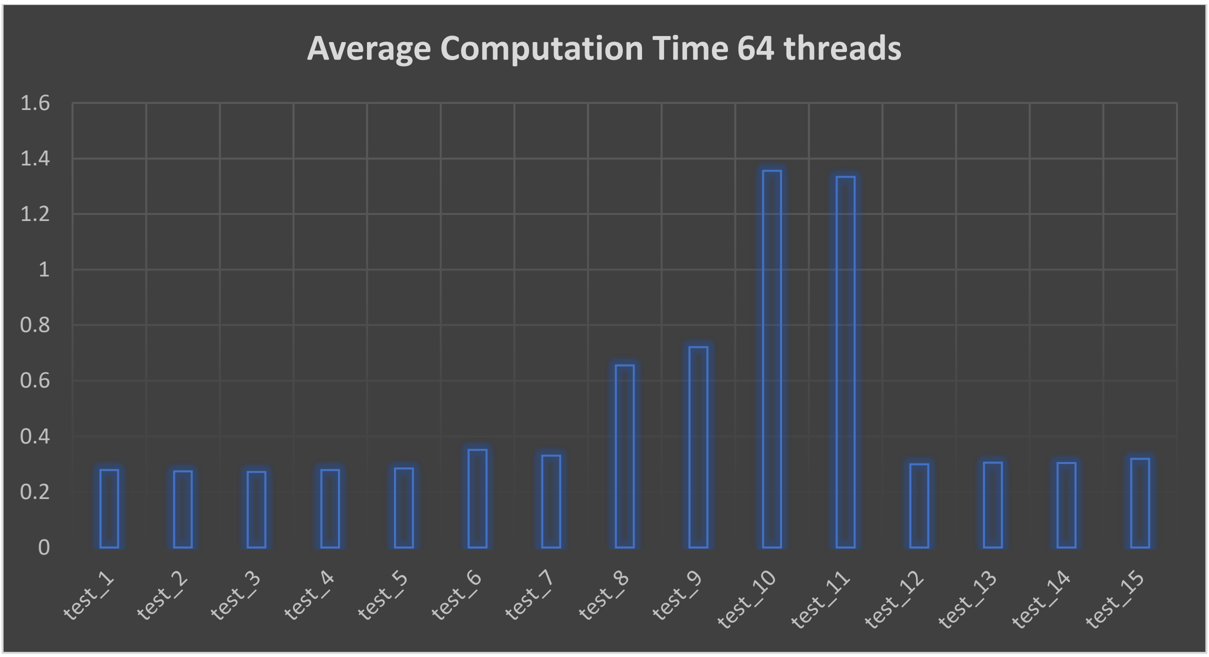
- **16-threaded execution:**



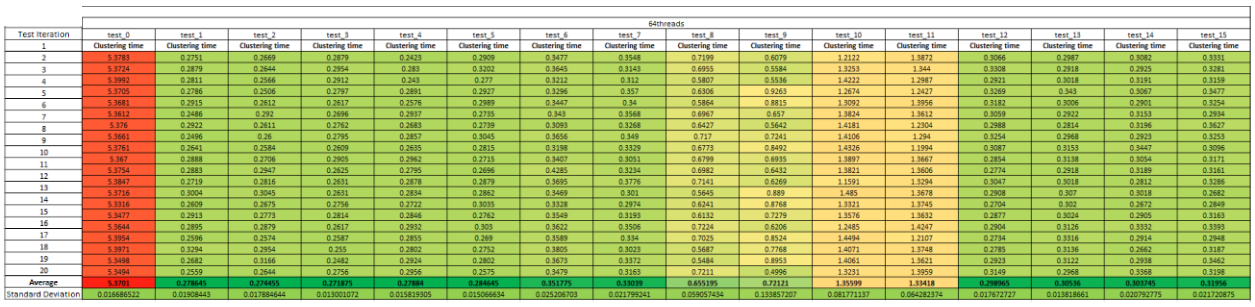
- **32-threaded execution:**



- **64-threaded execution:**



- **64-threaded execution results heatmap:**



	64threads															
Test Iteration	test_0	test_1	test_2	test_3	test_4	test_5	test_6	test_7	test_8	test_9	test_10	test_11	test_12	test_13	test_14	test_15
1	Clustering time	Clustering time	Clustering time	Clustering time	Clustering time	Clustering time	Clustering time	Clustering time	Clustering time	Clustering time	Clustering time	Clustering time	Clustering time	Clustering time	Clustering time	Clustering time
2	5.3783	0.2751	0.2669	0.2879	0.2423	0.2909	0.3477	0.3548	0.7199	0.6079	1.2122	1.3872	0.3066	0.2987	0.3082	0.3331
3	5.3724	0.2879	0.2644	0.2954	0.283	0.3202	0.3645	0.3143	0.6955	0.5584	1.3253	1.344	0.3308	0.2918	0.2925	0.3281
4	5.3992	0.2811	0.2566	0.2912	0.243	0.277	0.3212	0.312	0.5807	0.5536	1.4222	1.2987	0.2921	0.3018	0.3191	0.3159
5	5.3705	0.2786	0.2506	0.2797	0.2891	0.2927	0.3296	0.357	0.6306	0.9263	1.2674	1.2427	0.3269	0.343	0.3067	0.3477
6	5.3681	0.2915	0.2612	0.2617	0.2576	0.2989	0.3447	0.34	0.5864	0.8815	1.3092	1.3956	0.3182	0.3006	0.2901	0.3254
7	5.3612	0.2486	0.292	0.2696	0.2937	0.2735	0.343	0.3568	0.6967	0.657	1.3824	1.3612	0.3059	0.2922	0.3153	0.2934
8	5.376	0.2922	0.2611	0.2762	0.2683	0.2739	0.3093	0.3268	0.6427	0.5642	1.4181	1.2304	0.2988	0.2814	0.3196	0.3627
9	5.3661	0.2496	0.26	0.2795	0.2857	0.3045	0.3656	0.349	0.717	0.7241	1.4106	1.294	0.3254	0.2968	0.2923	0.3253
10	5.3761	0.2641	0.2584	0.2609	0.2635	0.2815	0.3198	0.3329	0.6773	0.8492	1.4326	1.1994	0.3087	0.3153	0.3447	0.3096
11	5.367	0.2888	0.2706	0.2905	0.2962	0.2715	0.3407	0.3051	0.6799	0.6935	1.3897	1.3667	0.2854	0.3138	0.3054	0.3171
12	5.3754	0.2883	0.2947	0.2625	0.2795	0.2696	0.4285	0.3234	0.6982	0.6432	1.3821	1.3606	0.2774	0.2918	0.3189	0.3161
13	5.3847	0.2719	0.2816	0.2631	0.2878	0.2879	0.3695	0.3776	0.7141	0.6269	1.1591	1.3294	0.3047	0.3018	0.2812	0.3286
14	5.3716	0.3004	0.3045	0.2631	0.2834	0.2862	0.3469	0.301	0.5645	0.889	1.485	1.3678	0.2908	0.307	0.3018	0.2682
15	5.3316	0.2609	0.2675	0.2756	0.2722	0.3035	0.3328	0.2974	0.6241	0.8768	1.3321	1.3745	0.2704	0.302	0.2672	0.2849
16	5.3477	0.2913	0.2773	0.2814	0.2846	0.2762	0.3549	0.3193	0.6132	0.7279	1.3576	1.3632	0.2877	0.3024	0.2905	0.3163
17	5.3644	0.2895	0.2879	0.2617	0.2932	0.303	0.3622	0.3506	0.7224	0.6206	1.2485	1.4247	0.2904	0.3126	0.3332	0.3393
18	5.3954	0.2596	0.2574	0.2587	0.2855	0.269	0.3589	0.334	0.7025	0.8524	1.4494	1.2107	0.2734	0.3316	0.2914	0.2948
19	5.3971	0.3294	0.2954	0.255	0.2802	0.2752	0.3805	0.3023	0.5687	0.7768	1.4071	1.3748	0.2785	0.3136	0.2662	0.3187
20	5.3498	0.2682	0.3166	0.2482	0.2924	0.2802	0.3673	0.3372	0.5484	0.8953	1.4061	1.3621	0.2923	0.3122	0.2938	0.3462
	5.3494	0.2559	0.2644	0.2756	0.2956	0.2575	0.3479	0.3163	0.7211	0.4996	1.3231	1.3959	0.3149	0.2968	0.3368	0.3198
Average	**5.3701**	**0.278645**	**0.274455**	**0.271875**	**0.27884**	**0.284645**	**0.351775**	**0.33039**	**0.655195**	**0.72121**	**1.35599**	**1.33418**	**0.298965**	**0.30536**	**0.303745**	**0.31956**
Standard Deviation	0.016686522	0.01908443	0.017884644	0.013001072	0.015819305	0.015066634	0.025206703	0.021799241	0.059057434	0.133857207	0.081771137	0.064282374	0.017672727	0.013818661	0.020792775	0.021720875

Literature

https://en.wikipedia.org/wiki/K-means_clustering