UNIVERSITÀ DEGLI STUDI
DI GENOVA

High Performance Computing (HPC) Project: Fuzzy C-Means Optimization Process

Andrea Canepa (S4185248)
Francesco Pellaco (S4228743)

AY 2019/2020

Contents

1 Introduction

As final HPC course project we decided to implement the **Fuzzy C-Means** algorithm and then to improve it in a *parallel* and *distributed* way. It is one of the most important *unsupervised learning* algorithms for ***clustering***, used for instance in *image segmentation* for *pattern recognition*, *object detection* and *medical imaging*. Since the focus of our project are the optimization techniques, we have generated some synthetic datasets with a small program written by us that extracts samples according to a Gaussian distribution. In particular most of the tests are performed with a set of 45000 points distributed on a 2D Cartesian plane over 9 centers.
Please notice that, since we would like to investigate on the performance of the algorithm itself, we ignored the time to read the dataset and to produce the output.
A small note on the algorithm: there are a lot of *variables* (e.g. the dataset, its size, ...), *free parameters* (e.g. number of clusters, fuzzyfication parameters, ...) and some *random values* (e.g. to choose the starting centroids, ...) that contribute to the final result; we take in account only a small subset of them during our experiments since the outcomes could change a lot with relatively small changes in those values.

For further references see also:

- `https://en.wikipedia.org/wiki/Fuzzy_clustering`
- `https://www.mathworks.com/help/fuzzy/fuzzy-c-means-clustering.html`

1.1 Project Structure

The project has the following structure:

- **common**: contains shared code parts between the files and also
  - **dataset**: contains our own generated datasets and the utility used to make them
  - **python_plot**: utilities written in Python3 to draw charts and result
- **configure.sh**: a configuration script to be run before performing the tests
- **cuda**: cuda optimized version of the code
- **do_tests.sh**: a script to perform batched tests on the server
- **Makefile**: an utility to compile the whole code with ease on the server
- **OpenMP**: parallel optimized version of the code
- **OpenMPI**: distributed version of the code
- **sequential**: vanilla version of the algorithm

1.2 Code and compilers

The project was coded in C++ standard ANSI version 17, with an extensive use of `#define` *macros* to speedup the compilation phase. The source code was compiled and linked with the Intel compiler for C++ `icpc` version 17.0.4, `mpicc` that relies on `gcc` version 4.8.5 and `nvcc` version 10.2.89. We produced different binaries changing compilers optimization flags, particularly we have used *-O0 -O2 -Ofast -xHost*. In the development stage we have used *-Wall -pedantic -ggdb -Werror* flags and, in order to produce a report, *-qopt-report=2(5) -qopt-report-phase=vec* too.

Below are listed the command lines we have used in order to compile the code:

```
    icpc -ansi -std=c++17 -DBIG sequential/*.cpp common/common.cpp
-O2 -o bin/sequential/fuzzycm_02

icpc -ansi -std=c++17 -DBIG OpenMP/*.cpp common/common.cpp -DSTATIC
-O2 -o bin/parallel/static/fuzzycm_static_02 -lm -qopenmp

mpiicc -ansi -std=c++17 -DBIG OpenMPI/*.cpp common/common.cpp -O2 -o
bin/mpi/fuzzycm_mpi_02

nvcc -gencode=arch=compute_30,code=sm_30 -I "C:\Program Files\NVIDIA
GPU Computing Toolkit\CUDA\v10.2\include" -I "C:\Program Files (x86)\Windows
Kits\10\Include\10.0.18362.0\ucrt" -I "C:\Program Files (x86)\Microsoft
Visual Studio\2019\Community\VC\Tools\MSVC\14.26.28801\include" --machine
64 -DMEDIUM -Xcompiler "/EHsc /W3 /Od /RTC1 /MDd" -o fuzzycmcu *.cu
```

While the latter was suggested by Visual Studio Compiler with Cuda integration, the other ones were written by us.

Figure 1: CPUs: Intel Xeon Phi 7210, AMD Phenom II X4 965 and i9-9880H

2 Hardware Architecture

The server where we have run the binaries on, with the exception of Cuda, is a cluster made up of 11 nodes, each with an ***Intel Xeon Phi 7210*** CPU.
Our Cuda system features an ***Nvidia GTX 770*** with *1536 Cuda cores* and an old ***AMD Phenom II X4 965 CPU***. As we can see from the `-gencode` parameter used for nvcc CUDA compilation our GPU family is identified as *30*. Every GPU architecture has its own code, in our case the architecture is ***Kepler***.
For the sake of testing we also decided to compile and run the sequential binary on the cluster *Phi CPU*, on the *AMD* one and on a new ***Intel i9-9880H***.

Figure 2: GPU we have used

3 Output

The algorithm generates a clustered result from the provided dataset as we can see by looking at the following output plotted with a `Python3` script while on the command-line console we can appreciate the time elapsed during the computation.
All the tests and results we will see in the report use the big dataset as input.

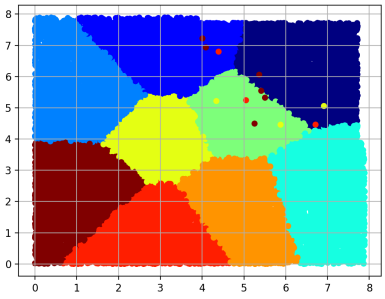

Figure 3: Big dataset result

4 Algorithm

The algorithm we present is a well-known clustering technique called ***Fuzzy C-Means***. For each point in the dataset it computes a *fuzzy indicator set*. Moreover, it uses a ***membership function*** to evaluate membership of samples to *clusters*. Membership of data point x_i to cluster with centroid y_j is a function of *Euclidean distance* $d(x_i, y_j)$. The whole algorithm relies on a ***fuzzyfication parameter***, m, which we decided to be 2. We defined also a *objective function* of cost to evaluate the convergence of the process, J. It follows a pseudo-code representation of the steps:

Input: X training set, c number of clusters, m fuzzyfication parameter, ϵ threshold, max_iter maximum number of iterations

1. initialize c centroids $y_1 \cdots y_c$ randomly 2. create an indicator vector u_i for each data point
3. iters := 0
4. WHILE iters < max_iter || $J < \epsilon$ DO
4.1. FOR EACH data point x_i DO
4.1.1. compute the distance $d_ij = \|x_i - y_j\|$ to each centroid y_j
4.1.2. compute the components $[u_{i1}, \cdots, u_{ic}]$ of the membership vector u_i as:

$$u_{ij} = \frac{1}{\sum_{k=1}^{c} (\frac{\|x_i - y_j\|}{\|x_i - y_k\|})^{\frac{2}{m-1}}}$$

4.2. compute each centroid y_j as:

$$y_j = \frac{\sum_{i=1}^{n} u_{ij} x_i}{\sum_j u_{ij}}$$

4.3. compute the objective function $J = \sum_i^n \sum_j^{\#C} u_{ij}^m \|x_i - c_j\|^2$

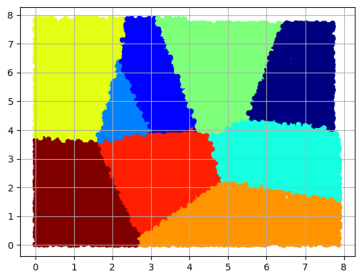

Figure 4: Example of execution with 9 clusters, $c = 9$

5 Sequential

The sequential algorithm is simply the base code, so in this case we did not make any optimization except for the compiler ones. In our tests we have used the `-O0, -O2, -Ofast` and `-xHost` flags to optimize the compilation.
In the following chart we can see how the non parallelized executable behaved on different CPUs.
We have to remember that when using `O0` the compiler does not optimize the code, however if we use `O2` it applies the level 2 of optimization where it optimizes both compilation time and execution. `Ofast`, instead, enables all the `O3` level optimizations and enables `-ffast-math` violating *IEEE* standards. Finally `xHost` is an Intel proprietary option which tells the compiler to generate instructions for the highest *instruction set* available on the compilation *host* processor, however this was set on the *Phi* cluster only where the *Intel* compiler (`icpc` for `C++`) was available.

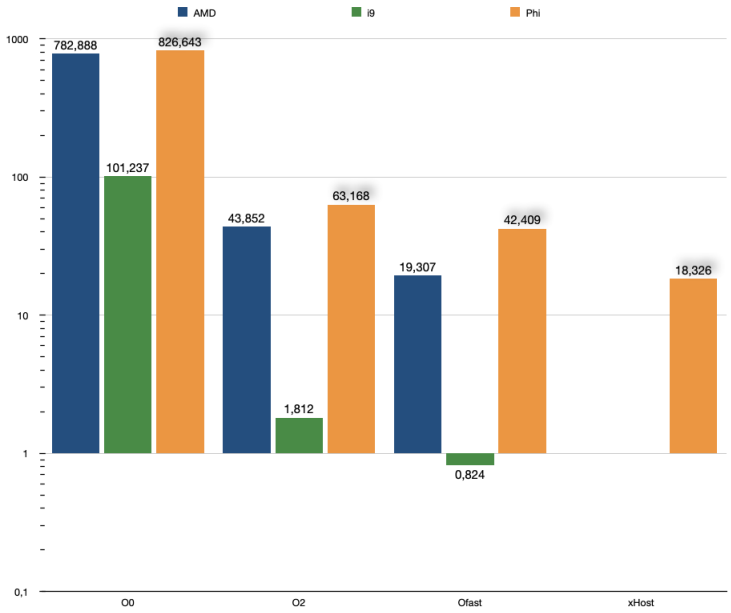

Figure 5: Sequential executable on different CPUs

Before proceeding with the optimization process we have generated and studied the report created by the Intel compiler providing it the arguments `-qopt-report=2` (also 5) and `-qopt-report-phase=vec`.

6 OpenMP

The first technique we applied to improve the performance of the algorithm makes use of OpenMP.
We targeted two functions: `fuzzyCmeans` and `objectiveFunction`. We tested all 3 different scheduling modes of execution, i.e. *static*, *dynamic* and *guided*. We expect that the **static** strategy will perform better than the others since the workload at each iteration is always the same. To produce a self-contained source file we have heavily used *#ifdef* preprocessor directives. An important observation is how we use the `firstprivate(·) #pragma directive` to guarantee that each thread has its own copy of the data but initialized from the outside since those are parameters taken in input from the exterior and they are not local variables. In the following we will show our results:

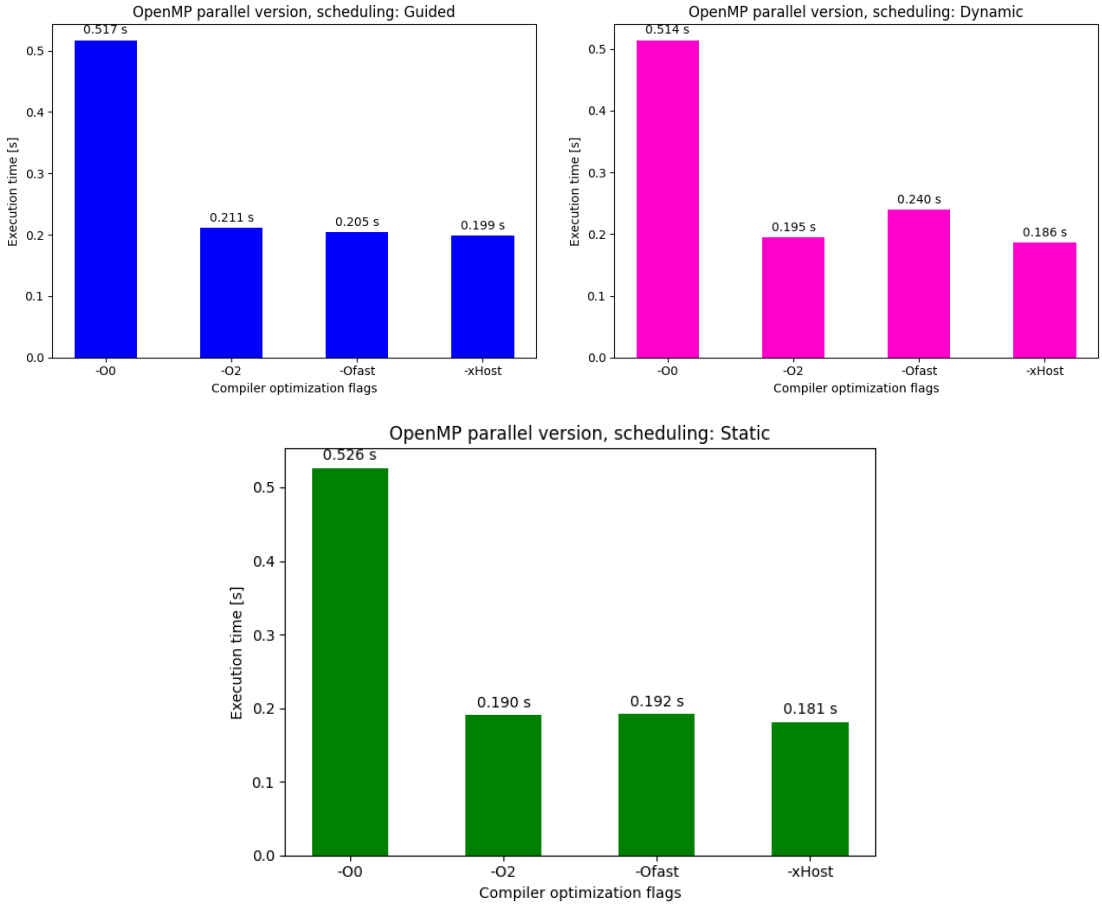

Figure 6: MP: Compilation flags comparison

To confirm once again our expectations we compare 10 executions of the same binary compiled with the `-xHost` flag since it is the one that guarantees us the best performance in terms of *execution time*.

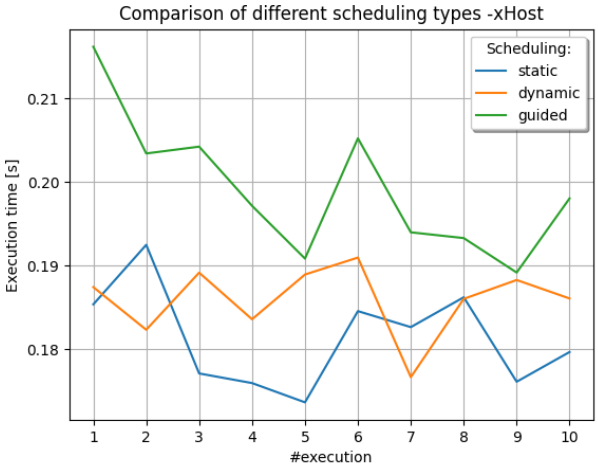

Figure 7: Scheduling techniques comparison

The *static* strategy is the one we should adopt, especially with very large datasets. Notice that we did not report the results obtained with smaller inputs since those outcomes were not reliable because of the *communication costs* between the threads generated by `OpenMP`. Also we tried to implement a `reduced` version but we omit it too for the same reason. To conclude this section, here some *statistics* of the execution times we have found.

```
*** SCHEDULING STRATEGY: Static ***
        BEST TIME : 0.174 s
       WORST TIME : 0.192 s
        AVG TIME : 0.181 s
*** SCHEDULING STRATEGY: Dynamic ***
        BEST TIME : 0.177 s
       WORST TIME : 0.191 s
        AVG TIME : 0.186 s
*** SCHEDULING STRATEGY: Guided ***
        BEST TIME : 0.189 s
       WORST TIME : 0.216 s
        AVG TIME : 0.199 s
```

7 OpenMPI

To implement this *distributed* version of our algorithm we have to face an interesting problem: we have to understand how to ***split data*** across the network of nodes and, in order to achieve this result, we have to rethink the implementation of some of the functions according to the **SPMD** (*Single Program Multiple Data*) model.

In this case we have to reshape `adjustClusterCenters(·)` and `objectiveFunction(·)` to achieve better performances. A noticeable point is that we have to split the data among the *nodes* present in the *cluster* keeping in mind that the better the points are divided, the better the outcomes will be (in terms of execution time). Each node has to deal with a $\frac{N}{P}$ fraction of the total number of points contained in the dataset, where N is the size of the training set and P is the number of processors in use. The trick is to rethink the `for` loops in such a way that each node has to elaborate only its portion of data, identified by a function of its *id* and the splitting fraction:

```
for (int i=myid*(SIZE/P); i<(myid+1)*(SIZE/P); i++)
```

Recall that we are executing our program over 11 different machines, $hpcocapie01, \cdots, hpcocapie11$, passing a file that contains their name as a parameter to `mpirun` with the switches `-nolocal --hostfile <file_with_hostnames>` which forces a distributed execution.
The only MPI operation that we need is `MPI_Allreduce` to combine values from all processes and to distribute the result back to all processes. We used it to perform different *sums*, passing the parameter `MPI_SUM`.

When we deal with OpenMPI it is interesting to study the **scaling** of the application. In particular, we are interested in two types of scaling:

- **strong**: fixed size of the problem, while augmenting computational power
- **weak**: fixed per-node problem size, while increasing both size and computational power

7.1 Strong scaling

We perform the experiment using a dataset with $N = 45000$, $c = 9$ and the "`-O2`" binary. Those are the results:

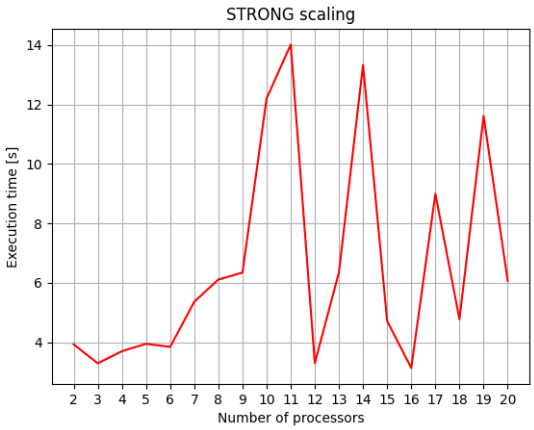

Figure 8: Strong scaling with OpenMPI

It shows a peculiar fluctuating behavior, but the best performance are obtained with, respectively, *3*, *12* and *16* processors. Do not forget that there is an implicit cost for the *communication* between processes (which is completely transparent to the final user) that has to be taken into account.

7.2 Speedup and Efficiency

In this scenario we can observe another 2 important measures of the effectiveness of the solution:

- **speedup**: $S_N(p) = \frac{T_N(1)}{T_N(p)}$
- **efficiency**: $E_N(p) = \frac{S_N(p)}{p}$

where p is the number of processors, N is the problem size and $T_N(\cdot)$ is the execution time with $\cdot$ processors.

Here are the results that reflect exactly what the *strong scaling* has forecast:

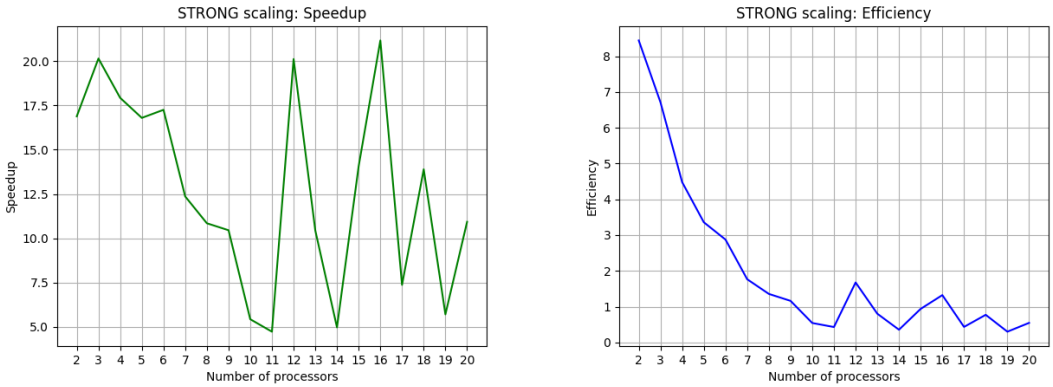

Figure 9: Speedup and Efficiency

7.3 Weak Scaling

To study the *weak scaling* trend we choose to distribute the load in such a way that each node has to process $\frac{1}{10}$ of the problem size, *1000 points/processor*. With less than 10 processors the execution was extremely inefficient, because of the extremely high *coordination* and *communication costs*. We can notice that the best configuration was with 12 processes distributed over the nodes, also because the dataset is split in a way that is very balanced among the workers.

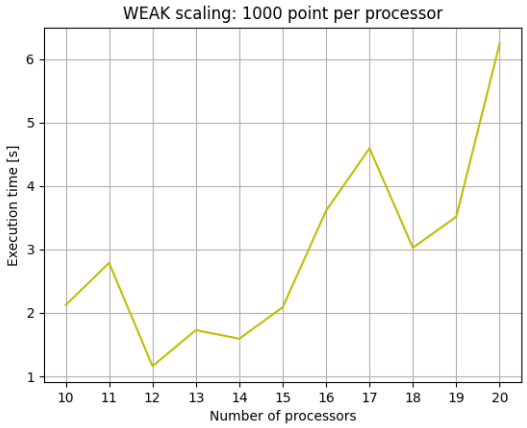

Figure 10: Weak scaling with OpenMPI

7.4 Performance

After all the previous experiments, we can state that with 12 processors it is possible to achieve the fastest execution. To test the optimization flags we decided to use this configuration.

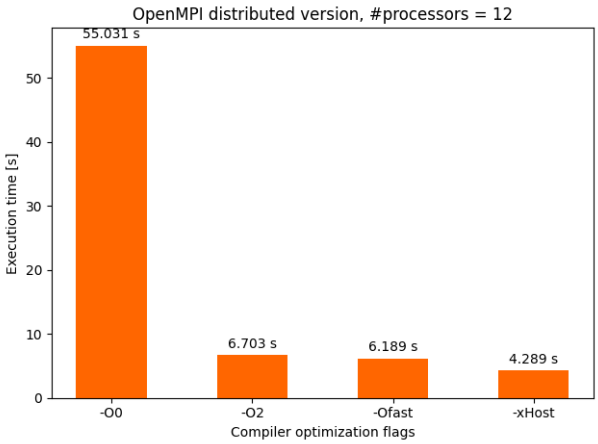

Figure 11: MPI: Compilation flags comparison

Notwithstanding we achieved a great improvement: the parallel version is much faster by an order of magnitude, we suggest to use that version if time is critical for your application.

8 CUDA

Finally our Cuda implementation managed to optimize the runtime by a lot, reducing it to around 2.5s. All we did was optimizing the `fuzzyCmeans(·)` (and so also `objectiveFunction(·)`) function by letting it run on the GPU Cuda cores (as we stated precedently 1536). We have also made the distance function shareable between the CPU and the GPU in order to declare just one function and to make it callable from both of them. In order to do so we simply added `__host__` and `__device__` next to the function declaration.
The following chart illustrates the runtime of the Cuda code per 10 executions.

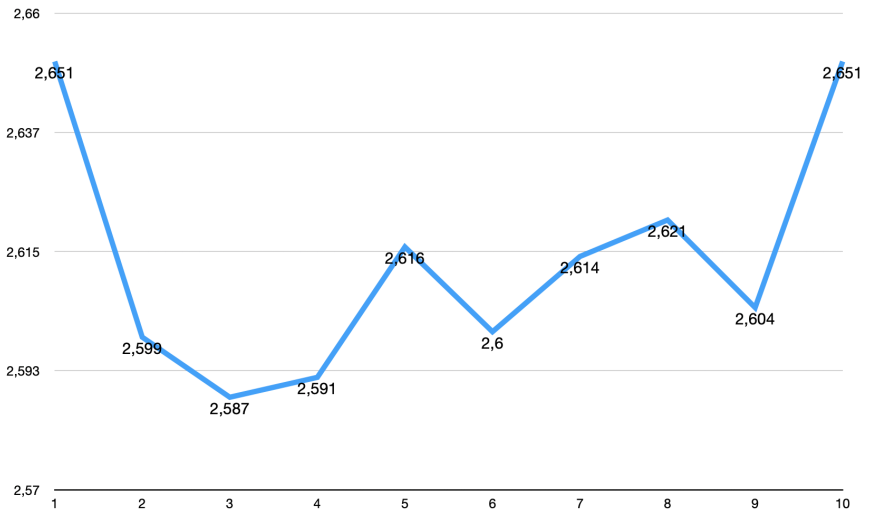

Figure 12: Chart of 10 executions

By looking at the chart we can clearly see that CUDA performs better than the sequential code, but also better than OpenMPI. However OpenMP manages to perform even better than CUDA, using any compiling parameter, even without any optimizations.