Project Genetics - OpenMP

Developing and Parallelizing a K-Means Clustering Algorithm

UPV-EHU - Computer Architecture
30 December 2022

Nicolás Aguado González, Nut Mora Guntín

Index

1. Introduction

The aim of this project is to develop an application using simplified machine learning and the OMP library. The goal is to classify genetic samples into different groups using the K-means clustering algorithm. We will be using the techniques that we have learned in the previous assignments to parallelize the application, taking into acount that some parts of the code don't benefit from parallelization. So, it's important to carefully analyze the provided code before we explain our program.

To understand the code we have first of all to understand its context. It's supposed that the World Health Organization (WHO) is conducting a genetic analysis of samples in its labs to better understand and prepare for pandemics like COVID-19. The WHO has a large database of genetic samples of pathogenic elements, each of which is identified by 40 genetic characteristics and is associated with a specific type of disease. The WHO has classified the diseases into 18 families of possible diseases.

We have been commissioned to process the database of genetic samples and classify each of them into its corresponding genetic group based on the similarity of its characteristics. For this, we will use the K-means clustering algorithm, which is a well-known method for unsupervised pattern classification.

The K-means algorithm works by dividing the input space into clusters, the goal is to create clusters where the objects within a cluster are similar, by minimizing the sum of squared errors among the elements of the same group. In our context, they will be grouped based on their closeness in terms of their characteristics, with those samples that have the highest similarities (or smallest distances) being placed into the same group.

2. Development of the project

Compilation and execution

We have different scripts to compile and execute the program, depending on the amount of elements `[#]` that we want to process, and if the execution is `serial` or `parallel` . Our (bash) scripts look roughly like this:

Name of the script: `ej[#][#]`

Contents of the script:

```
gcc -O2 -o [#]compilado[#] gengroups_[#].c fungg_[#].c -lm -fopenmp

./[#]compilado[#] ../ARC/Genetics/dbgen.dat ../ARC/Genetics/dbdise.dat
[#]
```

Now, when we want to execute a script, first we set the number of threads that we want to process the program with and then we have to use the terminal interpreter `bash` and pass it as a parameter the name of the script file (`ej, ejserial, ej1000, ej1000serial`):

```
[arc01@dif-cluster proy]$ export OMP_NUM_THREADS=32

[arc01@dif-cluster proy]$ bash ej[#][#]
```

To prove that our program works correctly we first execute the script and them compare the results obtained in the file `results_[#].out` and in the given file with the correct results. For example,

```
[arc01@dif-cluster proy]$ diff results_p.out results.out

102c102

<  Analysis of diseases (medians)

---

>  Analysis of deseases (medians)
```

We can see that both result files contain the same data. (Except for a typo). Which means that our program is working correctly, as it gives us the same results as the ones expected.

First steps - Code analysis

Our very first task was to understand the meaning of each struct/vector/matrix of the given code, and which information is actually stored in each of them.

The genetic samples will be clasificated in groups with the structure `ginfo`: `members` is a vector of the different pathogenic elements' indexes and `size` the number of elements in this vector.

```
#define MAXELE      230000 //number of elements (samples)
struct ginfo               // information about groups
{
 int  members[MAXELE];     // members
 int  size;                // number of elements
};
```

Next on, we get the `elems` matrix, that represents the pathogenic elements for each disease. In each row we'll get one pathogenic element, and each column will be each element's characteristic features.

```
#define NFEAT       40     //features of each instance
```

...

```
float            elems[MAXELE][NFEAT];
```

The `iingrs` vector will be used to store information about each group. Each element will be information about a group in the `ginfo` structure.

```
#define NGROUPSMAX  100    //number of clusters
```

...

```
struct ginfo     iingrs[NGROUPSMAX];
```

The `dise` matrix will contain probabilities for each element of developing each disease.

```
#define TDISEASE    18     //types of disease
```

...

```
float            dise[MAXELE][TDISEASE];
```

And, finally, the `disepro` vector will contain minimum and maximum probabilities amongst all groups of developing each disease.

```
struct analysis  disepro[TDISEASE];
```

Serial Part of the Development

Our second task was to complete the code to execute the serial version of the program. We will explain the different implementations that we've made of these different functions in `fungg_s.c` : `geneticdistance(...)`, `closestgroup(...)`, `validation(...)` and `diseases(...)`. And some additional auxiliary functions that we added to make the code easier to understand: `square(...)`, `max(...)` and `swap(...)`.

- square(...)

This simple method squares the input parameter. We've decided to use this auxiliary function and not use the `pow` function included in `<math.h>` because it's more efficient to just use this when calculating powers of two rather than use a more complex function just to do a simple multiplication.

```
double square(double a) {
  return a * a;
}
```

- max(...)

Given two float parameters, this function returns the maximum of both:

```
double max(float a, float b) {
  return (a > b) ? a : b;
}
```

- swap(...)

Given the addresses of two parameters, this method swaps the contents of both of them:

```
void swap(float *a, float *b) {
  float aux = *a;
  *a = *b;
  *b = aux;
}
```

- geneticdistance(...)

In this method, we will be using the Euclidean distance formula to calculate the distance between two elements:

$$\sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2}$$

We will receive as a parameter the genetic information of the elements (vectors of size 40), and we'll return the distance.

```
double geneticdistance (float *elem1, float *elem2)
{
  double res = 0.0;
  for (int i = 0; i < 40; i++) {
      res += square(elem1[i]-elem2[i]);
  }
  return sqrt(res);
}
```

- closestgroup(...)

For this procedure, we'll have to calculate the closest centroid for each element. We'll receive as input the number of elements (`nelems`) in the `elems` matrix, and the `elems, cent` matrixes. We'll have to output the closest group for each element in the `grind` vector. With the first loop (`i`) we'll select each element one by one, and then compare it to each centroid (second loop `j`). Once we get the minimum, we store the value in `grind` .

```
void closestgroup (int nelems, float elems[][NFEAT], float
cent[][NFEAT], int *grind)
{
  double mindistance, dist; int jmin, i, j;
  for (i = 0; i < nelems; i++) {
    mindistance = -1.0;
    jmin = 0; // just in case (it should never be 0)
    for (j = 0; j < ngroups; j++) {
      dist = geneticdistance(elems[i],cent[j]);
      if (mindistance == -1.0 || mindistance > dist) {
        mindistance = dist;   jmin = j;
      }
    }
    grind[i] = jmin;
  }
}
```

- validation(...)

With the validation function we will calculate the quality of the clusters produced by the K-means algorithm, getting the compactness of each cluster and the distance between the centroids of the different clusters.

For this, we have to initialize some variables:

- `res` : for intermediate values.
- `meangroupdist` : mean distance between pairs of elements within a single cluster.
- `meancentdist` : mean distance between the centroids of different clusters.
- `tempcvi` : value that will be used to calculate the final value of CVI (Cluster Validity Indexes);
- `op` : counter used to keep track of the number of pairs of elements that have been processed.

First, we are going to calculate the group compactness. For this, we have to iterate over each cluster ('`i`'), reset `res` and `meangroupdist` and iterate over each element of the cluster ('`j`'), where we calculate the distance between the element '`j`' and the following ones, iterating over '`k`' , using the `geneticdistance` function and adding the result to `res`. We will be counting with `op` the number of pairs of elements that we process. Once this process is over, we calculate the group compactness of the cluster in `meangroupdist` dividing `res` by `op` and we store this value in `compact[i]`.

Then, we get the cluster distances. For this, we reset the value of our auxiliary variable `res` and start iterating over each cluster ('`j`'), to calculate the distance between the centroids of the clusters ('`i`' and '`j`') using the `geneticdistance` function and adding the result to `res` again. After this, we calculate the mean distance between centroids of the different clusters by dividing `res/(ngroups-1)`.

After this process, in each cluster ('`j`') we calculate a temporary CVI, which will be the sumatory part of the formula:

$$CVI = \frac{1}{ngroups} \sum \frac{b(i) - a(i)}{max\{a(i), b(i)\}}$$

The final step is returning the final CVI, dividing the previous result by `ngroups`.

```
double validation (float elems[][NFEAT], struct ginfo *iingrs, float
cent[][NFEAT], float *compact)
{
  double res, meangroupdist, meancentdist, tempcvi;  int i, j, k, op;
  tempcvi = 0.0;
  for (i = 0; i < ngroups; i++) {
    res = 0.0; meangroupdist = 0.0; op = 0;
    // Calculate group compactness
    for (j = 0; j < iingrs[i].size; j++) {
      for (k = j+1; k < iingrs[i].size; k++) {
        res += geneticdistance(elems[iingrs[i].members[j]],
elems[iingrs[i].members[k]]);
        op++;
      }
    }
    if (op != 0) meangroupdist = res/op;
    compact[i] = meangroupdist; // compact == a
    // Calculate cluster distances
    res = 0.0;
    for (j = 0; j < ngroups; j++) {
      if (i != j) {
        res+= geneticdistance(cent[i], cent[j]);
      }
    }
    meancentdist = res/(ngroups-1); //meancentdist == b[i]
    // Calculate CVI index partial sum
    tempcvi += (meancentdist-meangroupdist)/max(meangroupdist,
meancentdist);

  }
  // Final CVI Calculation
  return tempcvi/ngroups;
}
```

- diseases(...)

The `diseases` function calculates the probabilities of developing certain disieses for each group of genetic samples.

For this, we initialize `median` that will be used to store the median of a group (result of dividing the size of the group by 2).

First of all, we iterate over each disease in `disepro`, to initialize `mmax` and `mmin`, those values are going to be probabilities, then they are going to be between 0 and 1, so we initialize them in `-1` and `2` setting the initial minimi=um and maximum probabilities for the disease to extreme values.

After this we can start calculating the probabilities for each disease (‘`i`’). We iterate over each group of samples (‘`j`’), if it’s not empty, for each element in the group (‘`k`’) we store the probability of developing the disease for the element in the `prob` array at index `k`. Then, we use the bubble sort algorithm to sort the `prob` array in ascending order, there we have used the `swap` auxiliary function to make the code easier swapping the probabilitier of elements `k` and `l`.

Once we have the probabilities we have to compare them with the `disepro` structure in order to establish the maximum and minimum probability. If the value of `prob[median]` is greater than the current `mmax` or smaller than `mmin` it will be updated as the new `mmax/mmin` and the index of the group `gmax/gmin` will store the new one (‘`j`’).

```
void diseases (struct ginfo *iingrs, float dise[][TDISEASE], struct
analysis *disepro)
{
  int i, j, k, l, median;
  for (i = 0; i < TDISEASE; i++) {
    disepro[i].mmax = -1;
    disepro[i].mmin = 2;
  }
  for (i = 0; i < TDISEASE; i++) {
    for (j = 0; j < ngroups; j++) {
      if (iingrs[j].size != 0) {
        float prob[iingrs[j].size];
        for(k = 0; k < iingrs[j].size; k++) {
          prob[k] = dise[iingrs[j].members[k]][i];
        }
        for (k = 0; k < iingrs[j].size-1; k++) {
          for (l = k + 1; l < iingrs[j].size; l++) {
            if (prob[k] > prob[l]) {
              swap(&prob[k], &prob[l]);
            }
          }
        }
        median = iingrs[j].size/2;
        if (disepro[i].mmax < prob[median]) {
          disepro[i].mmax = prob[median];
          disepro[i].gmax = j;
        }
        if (disepro[i].mmin > prob[median]) {
          disepro[i].mmin = prob[median];
          disepro[i].gmin = j;
        }
      }
    }
  }
}
```

3. Development of The Project

Parallel Part of the Development

Now, we'll parallelize the code to improve the execution time of our code.

Auxiliary Function Improvements

After some analysis, we've decided NOT to parallelize our auxiliary functions. That is because the calculations inside of them were so simple that it actually took longer for the program to open a parallel section than to actually perform the calculations.

The functions that we have not parallelized are:

- `double square(double a)`
- `double max(float a, float b)`
- `void swap(float *a, float *b)`
- `double geneticdistance (float *elem1, float *elem2)`
- `void firstcentroids (float cent[][NFEAT])`

Analysis Phase Improvements

- diseases(...)

In this function, we initially tried to parallelize the `for` loops inside the disease and group loops; thus, we tried to parallelize the actual processing. But, we ran into some problems: We could not parallelize the sorting algorithm and we measured times and it was actually slower to do it this way. We reasoned that because of the small size of `iingrs[j].size` it was actually not worth it to parallelize the inner code of the loops, and we focused our efforts in the outer loops. In the end, our code looked like this:

```
#pragma omp parallel for private(i,j,k,l, median) schedule(dynamic)
for (i = 0; i < TDISEASE; i++) {
  for (j = 0; j < ngroups; j++)
```

Even though `TDISEASE` < `NGROUPSMAX` (18 < 100) We decided to parallelize the `diseases` loop rather than the groups loop. That is, because in our analysis, it was actually faster. Furthermore, the amount of calculations are different for each disease, so we set the schedule to dynamic.

Clustering Phase Improvements

- validation(...)

The validation function calculates the compactness of each group. We've decided to parallelize it like this:

```
#pragma omp parallel for private(i,j,k,op, res, meangroupdist,
meancentdist) reduction(+:tempcvi) schedule(dynamic)
  for (i = 0; i < ngroups; i++) {
```

Our reasoning was that because `iingrs[i].size` will be maximum 100, and the calculations for each group are not that complex, we should parallelize the outer loop. Also, it would consume more time opening and closing parallel sections than the time it would spend actually doing the calculations.

Furthermore, the amount of calculations are different for each group, so we set the schedule to dynamic.

We could do it like this, because in the serial version, we didn't program three separate loops for each "phase" of the calculation, rather we did it all in a big loop. Also, that's why we'll need to have a reduction in the variable `tempcvi`.

- closestgroup(...)

For the function that determines the closest group, it was a matter of looking at the ranges of the loops. `MAXELE` > `NGROUPSMAX` , (230 000 > 100). It is evident that the `nelems` for loop does more iterations so we decided to parallelize it like this:

```
#pragma omp parallel private(i,j, mindistance, dist, jmin)
 {
   #pragma omp for schedule(dynamic)
   for (i = 0; i < nelems; i++)
```

Furthermore, the amount of calculations are different for each element, so we set the schedule to dynamic.

- newcentroids(...)

For this function we have opened different parallel sections.

The first loop resets the matrix to `additions` 0, as its indexes are defined by `ngroups` and `NFEAT+1`, it will be at most 100x41, so we only parallelize the groups' loop and we did it like this:

```
#pragma omp parallel for private(i, j) schedule(static)
  for (i=0; i<ngroups; i++) {
    for (j=0; j<NFEAT+1; j++) {
      additions[i][j] = 0.0;
    }
  }
```

Then, we have this other `for` that iterates over `nelems`, whose value might scale until 230000 elements (samples), so we must parallelize it, taking into account the reduction made over `additions`.

```
#pragma omp parallel for private(i, j) reduction (+: additions)
schedule(static)
  for (i=0; i<nelems; i++)
  {
    for (j=0; j<NFEAT; j++) {
      additions[grind[i]][j] += elems[i][j];
    }
    additions[grind[i]][NFEAT] ++;
  }
```

We also tried to parallelize the following `for` , but as we needed to put barriers in it, it wasn't giving us better times than the ones that we already have. That's the reason why we have decided not to parallelize it.

- gengroups_p.c (main)

For the main program, most parts can't be parallelized because it's mostly reading and writing on files. But, a small part in the section that evaluates the partition can actually be parallelized. This small part fills up the `iingrs` vector with values. The loop thart cycles through has a range of `MAXELE` (230 000), so it's very important that we do something about it. We've parallelized it like this:

```
#pragma omp parallel for private(group, count, i) shared(grind, iingrs)
schedule(dynamic)
    for (i=0; i<nelems; i++)
    {
      group = grind[i];
      #pragma omp critical (group)
      {
        count = iingrs[group].size;
        iingrs[group].members[count] = i;
        iingrs[group].size ++;
      }
    }
```

The `#pragma omp parallel for` divides the calculation for all the number of elements, but we may encounter problems when two cores access `iingrs[group]` at the same time. That's why we used a `#pragma omp critical` with the variable `group`.

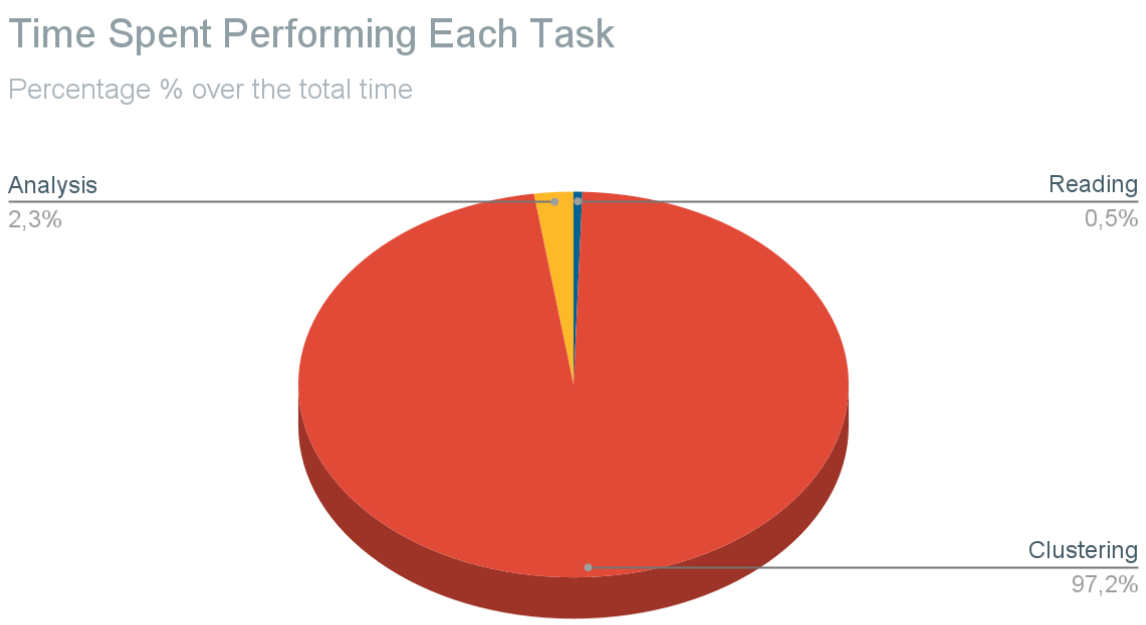
4. Conclusions

Serial Execution of the Program

When we run the K-Means Clustering Algorithm, we get really long execution times:

- To read the data the program takes `3,181` seconds
- To distribute the clusters the program takes `597,189` seconds
- In analyzing the data, the program spends `14,149` seconds
- And finally, in writing the data, the program takes `0,002` seconds.

In total, we've spent `614,521` seconds to get a result. Let's see the data as a graphic:



Now we can visualize it better, and understand what takes up most time in our program: and that is the clustering phase (97% !!). We can also see that the writing phase takes up a so small part of the execution that it doesn't even show up in the graphic.

So, looking at the results, we can conclude that we have to center our parallelizing efforts in the "clustering" phase, and just a little bit also in the "analysis" phase of our program.

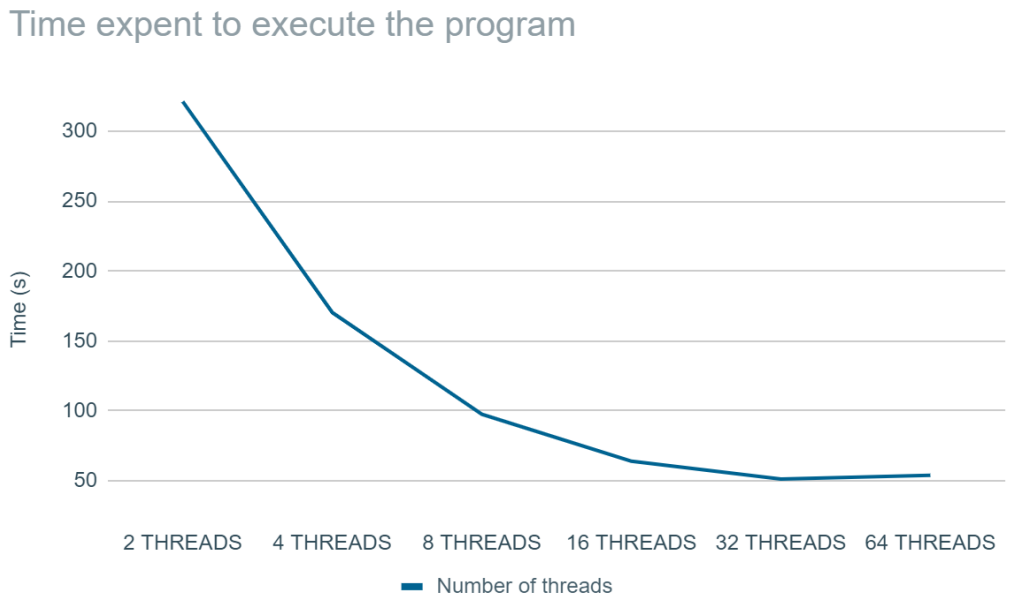
Parallel Execution and Analysis of the Program

Now that we have already parallelized our code, and we have also checked that it gives us the correct values, it's time to compare the time that it spends to execute the original serial program versus the parallelized one and see how much we have improved it.

When we run the K-Means Clustering Algorithm, we can see that the times that we get are really much smaller than the serial ones. We are going to analize the different execution times depending on the number of threads.

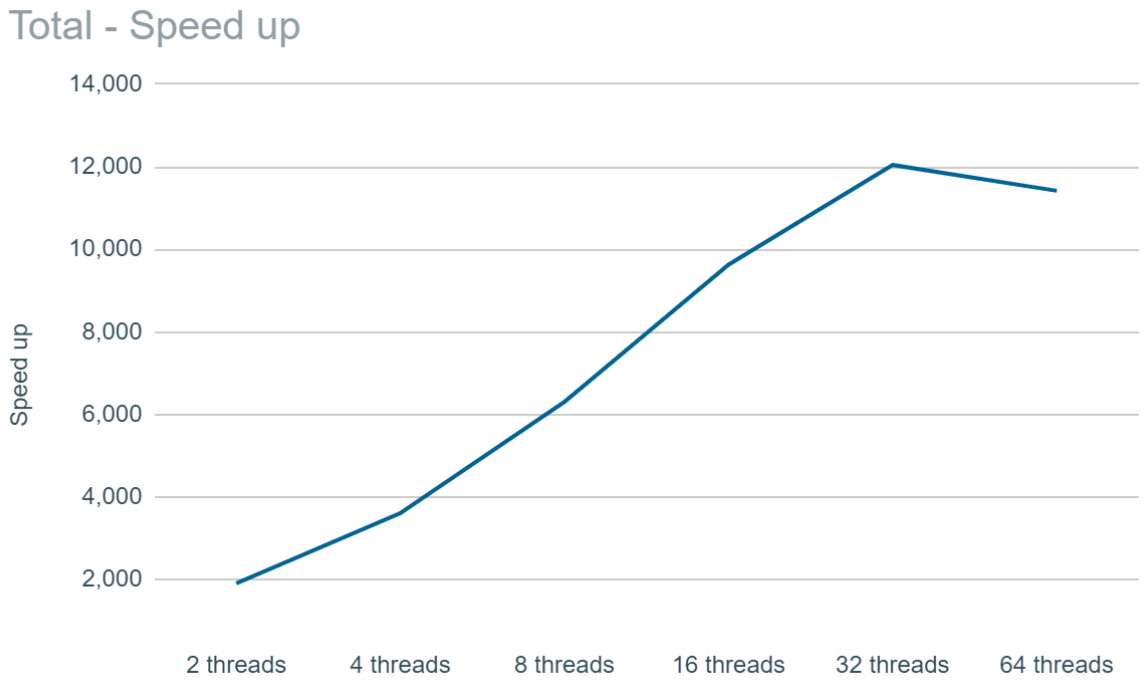
	2 THREADS	4 THREADS	8 THREADS	16 THREADS	32 THREADS	64 THREADS
Reading	3,285	3,287	3,288	3,285	3,287	3,281
Clustering	313,59	164,432	92,620	59,500	47,145	49,934
Analysis	4,494	2,494	1,506	1,016	0,583	0,587
Writing	0,002	0,002	0,002	0,002	0,003	0,003
Total	321,372	170,215	97,416	63,804	51,017	53,805

To make a first quick analysis with this data, we place the times in a graph in relation to the number of threads:

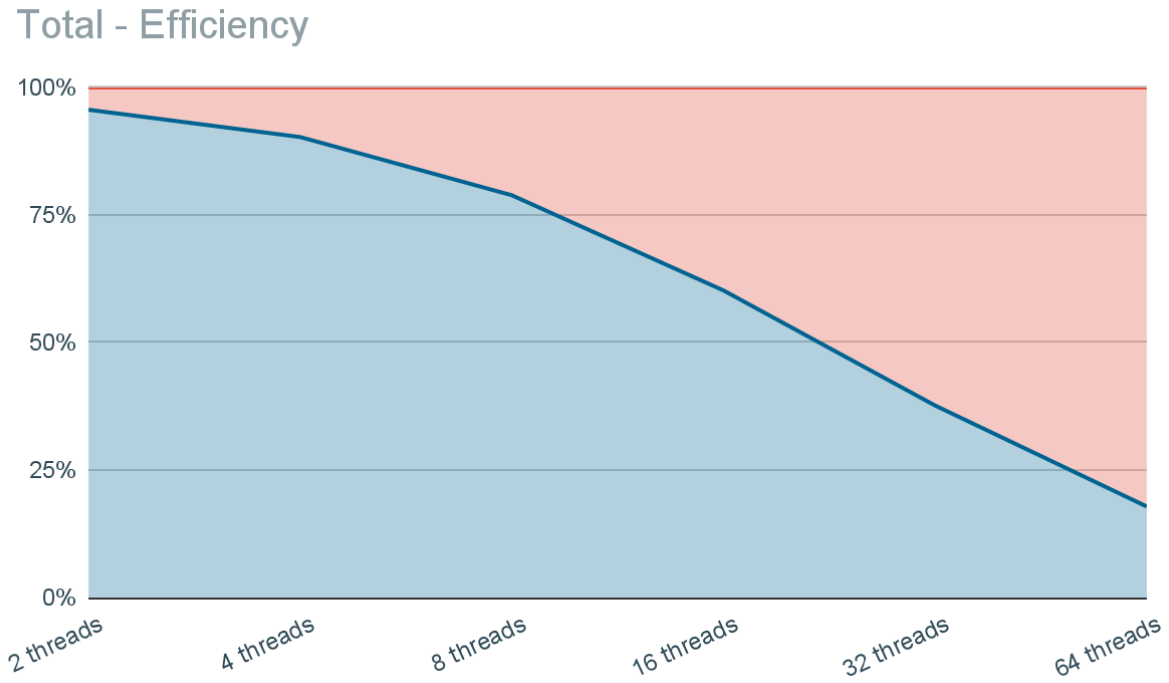


We can see how the times drop while we increase the number of threads, but this pattern suggests to us that the optimal number of threads is 32, as with 64 threads we have a slightly higher time.

The following graphic shows us the speed up (T_s/T_{sp}), where we compare the serial execution time with the one obtained in the parallel version. Now, we can see better how it raises until we reach 32 threads, which confirms our previous theory.



The last graphic, shows the efficiency ($T_{sp}/[\#]_{threads}$), a value between 0 and 100, which shows us the percentage of theoretical speed-up obtained with the different number of threads:



For a deeper analysis we could have taken into account the times of the different schedules (for each parallel region), but we've been doing it while programming. The reasoning in for each parallel region is explained in 'Parallel part of the development'. Overall, we can see that the analysis phase is completely dynamic, while in the clustering phase we can find two different schedules (static and dynamic).

In order not to be redundant showing our conclusions, we have chosen the clearest way to observe the data, which, taking into account that clustering represents almost all the execution time of the program, that is, observing directly the total amount of time expent within the different number of threads.

Like this, through our analysis of the total execution times for the parallel version of the program, we have determined that the optimal number of threads for the best performance is clearly 32 threads. This is due to the fact that beyond this number of threads, the overhead associated with managing and coordinating the work of additional threads begins to outweigh the benefits of a larger amount of threads in parallelization.

Additionally, we have to take into account that the effectiveness of parallelization may vary depending on the specific characteristics of the input samples data, as the clustering stage is the one that benefits the most from parallelization. But, all in all, the improvement we get when parallelizing the code is worth the effort, as it improves substancially.

5. Bibliography

- Slides present in https://egela.ehu.eus/ of the Computers Architecture Subject
- Errors in code and general troubleshooting https://stackoverflow.com/
- Book explanation of the K-Means Clustering:

  MacKay, David (2003). "Chapter 20. An Example Inference Task: Clustering" (PDF). *Information Theory, Inference and Learning Algorithms*. Cambridge University Press. pp. 284–292. ISBN 978-0-521-64298-9. MR 2012999.
- Online Manual for the gcc compiler and gdb debugger: https://gcc.gnu.org/onlinedocs/
- OpenMP API Documentation: https://www.openmp.org/specifications/
- How to create bash scripts and basic linux usage: https://howtogeek.com/
- C Language Documentation: https://learn.microsoft.com/
- Visual Studio Code and SSH Connections: https://code.visualstudio.com/docs/remote/
- Google Drive File Hosting and Document Editor Help: https://support.google.com/
- Cover photo for the report: https://istockphoto.com

6. Appendix

Complete Excel Analysis

[Google Drive download link]

Complete Serial Source Code

fungg_s.c

[Google Drive download link]

```
/*
   CA - OpenMP
   fungg_s.c
   Routines used in gengroups_s.c program
   Nicolás Aguado, Nut Mora - 30 December 2022
*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include "definegg.h"   // definition of constants

/* 0 - Function to calculate the square of a given double
*****************************************************************/
double square(double a) {
  return a * a;
}

/* 0 - Function to calculate the maximum out of two given doubles
*****************************************************************/
double max(float a, float b) {
  return (a > b) ? a : b;
}

/* 0 - Function to swap two variables
*****************************************************************/
void swap(float *a, float *b) {
  float aux = *a;
  *a = *b;
  *b = aux;
}
```

```
/* 1 - Function to calculate the genetic distance; Euclidean distance
between two elements.
       Input:   two elements of NFEAT characteristics (by reference)
       Output:  distance (double)
*****************************************************************/
double geneticdistance (float *elem1, float *elem2)
{
   // calculate the distance between two elements (Euclidean)
  double res = 0.0;
  for (int i = 0; i < 40; i++) {
      res += square(elem1[i]-elem2[i]);
  }
  return sqrt(res);
}

/* 2 - Function to calculate the closest group (closest centroid) for
each element.
   Input:   nelems   number of elements, int
            elems    matrix, with the information of the elements, of
size MAXELE x NFEAT, by reference
            cent    matrix, with the centroids, of size NGROUPS x
NFEAT, by reference
   Output:  grind   vector of size MAXELE, by reference, closest group
for each element
*****************************************************************/
void closestgroup (int nelems, float elems[][NFEAT], float
cent[][NFEAT], int *grind)
{
  // grind: closest group/centroid for each element
  double mindistance, dist; int jmin, i, j;
  for (i = 0; i < nelems; i++) {
    mindistance = -1.0;
    jmin = 0; // por si acaso se pone tonto, nunca deberia ser 0
    for (j = 0; j < ngroups; j++) {
      dist = geneticdistance(elems[i],cent[j]);
      if (mindistance == -1.0 || mindistance > dist) {
        mindistance = dist;
        jmin = j;
      }
    }
    grind[i] = jmin;
  }
}
```

```
/* 3 - Function to validate the classification: group compactness and
centroids compactness
       Calculate the CVI index
       Input:  elems    elements (matrix of size MAXELE x NFEAT, by
reference)
               iingrs   indices of the elements in each group (vector
with information for each group)
               cent     matrix, with the centroids, by reference
       Output: cvi index
               compact  compactness of each group (vector of size
NGROUPS, by reference)
*****************************************************************/
double validation (float elems[][NFEAT], struct ginfo *iingrs, float
cent[][NFEAT], float *compact)
{
  double res, meangroupdist, meancentdist, tempcvi;  int i, j, k, op;
  tempcvi = 0.0;
  for (i = 0; i < ngroups; i++) {
    res = 0.0; meangroupdist = 0.0; op = 0;
    for (j = 0; j < iingrs[i].size; j++) { // iingrs[i].size is the num
of groups
      for (k = j+1; k < iingrs[i].size; k++) {
        res += geneticdistance(elems[iingrs[i].members[j]],
elems[iingrs[i].members[k]]);
        op++;
      }
    }
    // if cluster is empty compactness is 0
    if (op != 0) meangroupdist = res/op;
    compact[i] = meangroupdist; // compact == a
    res = 0.0;
    for (j = 0; j < ngroups; j++) {
      if (i != j) {
        res+= geneticdistance(cent[i], cent[j]);
      }
    }
    meancentdist = res/(ngroups-1);
    tempcvi += (meancentdist-meangroupdist)/max(meangroupdist,
meancentdist);

  }
  return tempcvi/ngroups;
}
```

```
/* 4 - Function to analyse diseases
   Input:  iingrs   indices of the elements in each group (matrix with
MAXELE elements per group, by reference)
           dise     information about the diseases
   Output: disepro  analysis of the diseases: maximum, minimum of the
medians and groups
*****************************************************************
*****************************/
void diseases (struct ginfo *iingrs, float dise[][TDISEASE], struct
analysis *disepro)
{
  int i, j, k, l, median;
  for (i = 0; i < TDISEASE; i++) {
    disepro[i].mmax = -1;
    disepro[i].mmin = 2;
  }
  for (i = 0; i < TDISEASE; i++) {
    for (j = 0; j < ngroups; j++) {
      if (iingrs[j].size != 0) {
        float prob[iingrs[j].size];
        for(k = 0; k < iingrs[j].size; k++) {
          prob[k] = dise[iingrs[j].members[k]][i];
        }
        for (k = 0; k < iingrs[j].size-1; k++) {
          for (l = k + 1; l < iingrs[j].size; l++) {
            if (prob[k] > prob[l]) {
              swap(&prob[k], &prob[l]);
            }
          }
        }
        median = iingrs[j].size/2;
        if (disepro[i].mmax < prob[median]) {
          disepro[i].mmax = prob[median];
          disepro[i].gmax = j;
        }
        if (disepro[i].mmin > prob[median]) {
          disepro[i].mmin = prob[median];
          disepro[i].gmin = j;
        }
      }
    }
  }
}
```

```
// TWO OTHER FUNCTIONS IN THE MAIN PROGRAM
// =======================================

/* 5 - Initial values for centroids
************************************************************/
void firstcentroids (float cent[][NFEAT])
{
  int  i, j;
  srand (147);
  for (i=0; i<ngroups; i++)
  for (j=0; j<NFEAT/2; j++)
  {
     cent[i][j] = (rand() % 10000) / 100.0;
     cent[i][j+NFEAT/2] = cent[i][j];
  }
}

/* 6 - New centroids
************************************************************/
int newcentroids (float elems[][NFEAT], float cent[][NFEAT], int
grind[], int nelems)
{
  int     i, j, finish;
  float   newcent[ngroups][NFEAT];
  double  discent;
  double  additions[ngroups][NFEAT+1];

  // calculate new centroids for each group:  average of each dimension
or feature
  // additions: to accumulate the values for each feature and cluster.
Last value: number of elements in the group

  for (i=0; i<ngroups; i++)
  for (j=0; j<NFEAT+1; j++)
    additions[i][j] = 0.0;

  for (i=0; i<nelems; i++)
  {
    for (j=0; j<NFEAT; j++)
      additions[grind[i]][j] += elems[i][j];
    additions[grind[i]][NFEAT] ++;
  }
```

```
  finish = 1;
  for (i=0; i<ngroups; i++)
  {
    if (additions[i][NFEAT] > 0) //the group is not empty
    {
      for (j=0; j<NFEAT; j++) newcent[i][j] = additions[i][j] /
additions[i][NFEAT];

      // decide if the process needs to be finished

      discent = geneticdistance (&newcent[i][0], &cent[i][0]);
      if (discent > DELTA1) finish = 0;         // there is change at
least in one of the dimensions; continue with the process

      // copy new centroids
      for (j=0; j<NFEAT; j++) cent[i][j] = newcent[i][j];
     }
   }

   return (finish);
}
```

Complete Parallel Source Code

fungg_p.c

[Google Drive download link]

```
/*
   CA - OpenMP
   fungg_p.c
   Routines used in gengroups_p.c program
   Nicolás Aguado, Nut Mora - 30 December 2022
*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include <omp.h>
#include "definegg.h"   // definition of constants

/* 0 - Function to calculate the square of a given double
       Input : One double
       Output : The square of the double (a*a)
*****************************************************************/
double square(double a) {
  return a * a;
}

/* 0 - Function to calculate the maximum out of two given doubles
       Input : Two doubles
       Output : The maximum of both
*****************************************************************/
double max(float a, float b) {
  return (a > b) ? a : b;
}

/* 0 - Function to swap two variables
       Input : Two variables
       Output : Both of them, swapped
*****************************************************************/
void swap(float *a, float *b) {
  float aux = *a;
  *a = *b;
  *b = aux;
}
```

```
/* 1 - Function to calculate the genetic distance; Euclidean distance
between two elements.
*****************************************************************/
double geneticdistance (float *elem1, float *elem2) // no paralelizar?
{
  double res = 0.0;
  for (int i = 0; i < 40; i++) {
      res += square(elem1[i]-elem2[i]);
  }
  return sqrt(res);
}

/* 2 - Function to calculate the closest group (closest centroid) for
each element.
*****************************************************************/
void closestgroup (int nelems, float elems[][NFEAT], float
cent[][NFEAT], int *grind)
{
  // grind: closest group/centroid for each element
  double mindistance = DBL_MAX, dist; int jmin, i, j;
  #pragma omp parallel private(i,j, mindistance, dist, jmin)
shared(elems, cent, grind)
  {
    #pragma omp for schedule(dynamic) // EASY PARALLEL 230000 > 100
    for (i = 0; i < nelems; i++) { // MAXELE 230000 (paralelizar)
      mindistance = -1.0;
      jmin = 0; // por si acaso se pone tonto, nunca deberia ser 0
      for (j = 0; j < ngroups; j++) { // MAX 100
        dist = geneticdistance(elems[i],cent[j]);
        if (mindistance == -1.0 || mindistance > dist) {
          mindistance = dist;
          jmin = j;
        }
      }
      grind[i] = jmin;
    }
  }
}


/* 3 - Function to validate the classification: group compactness and
centroids compactness
*****************************************************************/
```

```
double validation (float elems[][NFEAT], struct ginfo *iingrs, float
cent[][NFEAT], float *compact)
{
  double res, meangroupdist, meancentdist, tempcvi;  int i, j, k, op;
  tempcvi = 0.0;
  #pragma omp parallel for private(i,j,k,op, res, meangroupdist,
meancentdist) reduction(+:tempcvi) schedule(dynamic)
  for (i = 0; i < ngroups; i++) {
    res = 0.0; meangroupdist = 0.0; op = 0;
    for (j = 0; j < iingrs[i].size; j++) { // iingrs[i].size is the num
of groups
      for (k = j+1; k < iingrs[i].size; k++) {
        res += geneticdistance(elems[iingrs[i].members[j]],
elems[iingrs[i].members[k]]);
        op++;
      }
    }
    if (op != 0) meangroupdist = res/op;
    compact[i] = meangroupdist; // compact == a
    res = 0.0;
    for (j = 0; j < ngroups; j++) {
      if (i != j) {
        res+= geneticdistance(cent[i], cent[j]);
      }
    }
    meancentdist = res/(ngroups-1);
    tempcvi += (meancentdist-meangroupdist)/max(meangroupdist,
meancentdist);
  }
  return tempcvi/ngroups;
}

/* 4 - Function to analyse diseases
*****************************************************************/
void diseases (struct ginfo *iingrs, float dise[][TDISEASE], struct
analysis *disepro)
{
  int i, j, k, l, median;
  for (i = 0; i < TDISEASE; i++) {
    disepro[i].mmax = -1;
    disepro[i].mmin = 2;
  }
```

```
  #pragma omp parallel for private(i,j,k,l, median) schedule(dynamic)
  for (i = 0; i < TDISEASE; i++) {
    for (j = 0; j < ngroups; j++) {
      if (iingrs[j].size != 0) {
        float prob[iingrs[j].size];
        for(k = 0; k < iingrs[j].size; k++) {
          prob[k] = dise[iingrs[j].members[k]][i];
        }

        for (k = 0; k < iingrs[j].size-1; k++) {
          for (l = k + 1; l < iingrs[j].size; l++) {
            if (prob[k] > prob[l]) {
              swap(&prob[k], &prob[l]);
            }
          }
        }
        median = iingrs[j].size/2;
        if (disepro[i].mmax < prob[median]) {
          disepro[i].mmax = prob[median];
          disepro[i].gmax = j;
        }
        if (disepro[i].mmin > prob[median]) {
          disepro[i].mmin = prob[median];
          disepro[i].gmin = j;
        }
      }
    }
  }
}

/* 5 - Initial values for centroids
************************************************************/
void firstcentroids (float cent[][NFEAT])
{
  int  i, j;
  srand (147);
  for (i=0; i<ngroups; i++)
  for (j=0; j<NFEAT/2; j++)
  {
     cent[i][j] = (rand() % 10000) / 100.0;
     cent[i][j+NFEAT/2] = cent[i][j];
  }
}
```

```
/* 6 - New centroids
*****************************************************************/
int newcentroids (float elems[][NFEAT], float cent[][NFEAT], int
grind[], int nelems)
{
  int     i, j, finish;
  float   newcent[ngroups][NFEAT];
  double  discent;
  double  additions[ngroups][NFEAT+1];

  #pragma omp parallel for private(i, j) schedule(static)
  for (i=0; i<ngroups; i++) {
    for (j=0; j<NFEAT+1; j++) {
      additions[i][j] = 0.0;
    }
  }

  #pragma omp parallel for private(i, j) reduction (+: additions)
schedule(static)
  for (i=0; i<nelems; i++)
  {
    for (j=0; j<NFEAT; j++) {
      additions[grind[i]][j] += elems[i][j];
    }
    additions[grind[i]][NFEAT] ++;
  }
  finish = 1;
  for (i=0; i<ngroups; i++)
  {
    if (additions[i][NFEAT] > 0) //the group is not empty
    {
      for (j=0; j<NFEAT; j++) newcent[i][j] = additions[i][j] /
additions[i][NFEAT];
      // decide if the process needs to be finished
      discent = geneticdistance (&newcent[i][0], &cent[i][0]);
      if (discent > DELTA1) finish = 0;        // there is change at
least in one of the dimensions; continue with the process
      // copy new centroids
      for (j=0; j<NFEAT; j++) cent[i][j] = newcent[i][j];
     }
   }
   return (finish);
}
```

gengroups_p.c

[Google Drive download link]

```
/*
    CA - practical work OpenMP
    gengroups_p.c SERIAL VERSION

    Processing genetic characteristics to discover information about
diseases
    Classify in NGROUPS groups, elements of NFEAT features, according
to "distances"

    Input:  dbgen.dat      input file with genetic information
            dbdise.dat     input file with information about diseases
    Output: results_s.out  centroids, number of group members and
compactness, and diseases

    Compile with module fungg_p.c and include option -lm
*****************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

#include "definegg.h"
#include "fungg.h"

float            elems[MAXELE][NFEAT];  // matrix to keep information
about every element
struct ginfo     iingrs[NGROUPSMAX];  // vector to store information
about each group: members and size

float            dise[MAXELE][TDISEASE];// probabilities of diseases
(from dbdise.dat)
struct analysis  disepro[TDISEASE]; // vector to store information
about each disease (max, min, group...)

int  ngroups = 35;      // initial number of groups

// Main program
// ============
```

```
void main (int argc, char *argv[])
{
  float   cent[NGROUPSMAX][NFEAT], newcent[NGROUPSMAX][NFEAT];
//centroid and new centroid
  float   compact[NGROUPSMAX];    // compactness of each group or
cluster

  int     i, j, nelems, group, count;
  int     grind[MAXELE];      //group assigned to each element
  int     finish = 0, niter = 0, finish_classif;
  double  cvi, cvi_old, diff;

  FILE    *f1, *f2;
  struct timespec  t1, t2, t3, t4, t5;
  double  t_read, t_clus, t_anal, t_write;

  if ((argc < 3) || (argc > 4)) {
    printf ("ATTENTION:  progr file1 (elems) file2 (dise) [num
elems])\n");
    exit (-1);
  }

  printf ("\n >> PARALLEL execution\n");
  clock_gettime (CLOCK_REALTIME, &t1);

  // read data from files: elems[i][j] and dise[i][j]
  // ================================================
  f1 = fopen (argv[1], "r");
  if (f1 == NULL) {
    printf ("Error opening file %s \n", argv[1]);
    exit (-1);
  }

  fscanf (f1, "%d", &nelems);
  if (argc == 4) nelems = atoi(argv[3]);

  for (i=0; i<nelems; i++)
  for (j=0; j<NFEAT; j++)
    fscanf (f1, "%f", &(elems[i][j]));

  fclose (f1);

  f1 = fopen (argv[2], "r");
```

```
  if (f1 == NULL) {
    printf ("Error opening file %s \n", argv[1]);
    exit (-1);
  }

  for (i=0; i<nelems; i++)
  for (j=0; j<TDISEASE; j++)
    fscanf (f1, "%f", &dise[i][j]);

  fclose (f1);

  clock_gettime (CLOCK_REALTIME, &t2);

  // PHASE 1: iterative process to classify elements in ngroups groups
  // until a minimum difference (DELTA2) in the CVI index
  // ================================================================

  finish_classif = 0;
  cvi_old = -1;

  while ((ngroups < NGROUPSMAX) && (finish_classif == 0))
  {
    // select randomly the first centroids
    firstcentroids (cent);

    // A. Classification process, ngroups
    // ==================================
    niter = 0;
    finish = 0;
    while ((finish == 0) && (niter < MAXIT))
    {
      // Obtain the closest group or cluster for each element
      closestgroup (nelems, elems, cent, grind);

      // Calculate new centroids and decide to finish or not depending
on DELTA
      finish = newcentroids (elems, cent, grind, nelems);
      niter ++;
    }

    // B. Evaluation of the partition
    // ==============================
    for (i=0; i<ngroups; i++) iingrs[i].size = 0;
```

```
    // number of elements and elements of each group
    #pragma omp parallel for private(group, count, i) shared(grind,
iingrs) schedule(dynamic)
    for (i=0; i<nelems; i++)
    {
      group = grind[i];
      #pragma omp critical (group)
      {
        count = iingrs[group].size;
        iingrs[group].members[count] = i;
        iingrs[group].size ++;
      }
    }

    // validation process and convergence
    cvi = validation (elems, iingrs, cent, compact);

    diff = cvi - cvi_old;
    if (diff < DELTA2) finish_classif = 1;
    else {
       ngroups += 10;
       cvi_old = cvi;
     }
   }

   clock_gettime (CLOCK_REALTIME, &t3);

  // PHASE 2: analyse diseases
  // =========================
    diseases (iingrs, dise, disepro);
    clock_gettime (CLOCK_REALTIME, &t4);

  // write results in a file
  // =======================
  f2 = fopen ("results_p.out", "w");
  if (f2 == NULL) {
    printf ("Error when opening file results_p.outs \n");
    exit (-1);
  }

  fprintf (f2, " >> Centroids of groups \n\n");
  for (i=0; i<ngroups; i++) {
```

```
    for (j=0; j<NFEAT; j++) fprintf (f2, "%7.3f", cent[i][j]);
    fprintf (f2,"\n");
  }

  fprintf (f2, "\n >> Size of the groups: %d groups \n\n", ngroups);
  for (i=0; i<ngroups/10; i++) {
    for (j=0; j<10; j++) fprintf (f2, "%9d", iingrs[10*i+j].size);
    fprintf(f2, "\n");
  }
  for (i=i*10; i<ngroups; i++) fprintf (f2, "%9d", iingrs[i].size);
  fprintf (f2, "\n");

  fprintf (f2, "\n >> Group compactness \n\n");
  for (i=0; i<ngroups/10; i++) {
    for (j=0; j<10; j++) fprintf (f2, "%9.2f", compact[10*i+j]);
    fprintf(f2, "\n");
  }
  for (i=i*10; i<ngroups; i++) fprintf (f2, "%9.2f", compact[i]);
  fprintf (f2, "\n");

  fprintf (f2, "\n\n Analysis of diseases (medians)\n\n");
  fprintf (f2, "\n Dise.  M_max - Group   M_min - Group");
  fprintf (f2, "\n ==================================\n");
  for (i=0; i<TDISEASE; i++)
    fprintf (f2, "  %2d     %4.2f - %2d      %4.2f - %2d\n", i,
disepro[i].mmax,
      disepro[i].gmax, disepro[i].mmin, disepro[i].gmin);

  fclose (f2);

  clock_gettime (CLOCK_REALTIME, &t5);

  // print some results in the screen
  // ================================
  t_read    = (t2.tv_sec-t1.tv_sec) + (t2.tv_nsec-t1.tv_nsec) /
(double)1e9;
  t_clus    = (t3.tv_sec-t2.tv_sec) + (t3.tv_nsec-t2.tv_nsec) /
(double)1e9;
  t_anal    = (t4.tv_sec-t3.tv_sec) + (t4.tv_nsec-t3.tv_nsec) /
(double)1e9;
  t_write   = (t5.tv_sec-t4.tv_sec) + (t5.tv_nsec-t4.tv_nsec) /
(double)1e9;
```

```
  printf ("\n    Number of iterations: %d", niter);
  printf ("\n    T_read:    %6.3f s", t_read);
  printf ("\n    T_clus:    %6.3f s", t_clus);
  printf ("\n    T_anal:    %6.3f s", t_anal);
  printf ("\n    T_write:   %6.3f s", t_write);
  printf ("\n    ========================");
  printf ("\n    T_total:  %6.3f s\n\n", t_read + t_clus + t_anal +
t_write);

  printf ("\n >> Centroids 0, 30 and 60 \n ");
  for (j=0; j<NFEAT; j++) printf ("%7.3f", cent[0][j]);
  printf("\n");
  for (j=0; j<NFEAT; j++) printf ("%7.3f", cent[20][j]);
  printf("\n");
  for (j=0; j<NFEAT; j++) printf ("%7.3f", cent[40][j]);
  printf("\n");

  printf ("\n >> Size of the groups: %d groups \n\n", ngroups);
  for (i=0; i<ngroups/10; i++) {
    for (j=0; j<10; j++) printf ("%9d", iingrs[10*i+j].size);
    printf("\n");
  }
  for (i=i*10; i<ngroups; i++) printf ("%9d", iingrs[i].size);
  printf ("\n");

  printf ("\n >> Group compactness \n\n");
  for (i=0; i<ngroups/10; i++) {
    for (j=0; j<10; j++) printf ("%9.2f", compact[10*i+j]);
    printf("\n");
  }
  for (i=i*10; i<ngroups; i++) printf ("%9.2f", compact[i]);
  printf ("\n");

  printf ("\n\n Analysis of diseases (medians)\n\n");
  printf ("\n Dise.  M_max - Group   M_min - Group");
  printf ("\n ==================================\n");
  for (i=0; i<TDISEASE; i++)
    printf ("  %2d     %4.2f - %2d      %4.2f - %2d\n", i,
disepro[i].mmax,
      disepro[i].gmax, disepro[i].mmin, disepro[i].gmin);

  printf("\n");
}
```