Faculty of Informatics

UPV/EHU

Informatics Engineering Degree

Computer Architecture

Computer Architecture and Technology Department

Multiprocessor systems: OpenMP practical work

Multiprocessor systems. Practical work. Parallelisation of an application using OpenMP.

In the third topic of the subject, we are analysing the bases of multiprocessor systems and, in order to program shared memory applications, we are using OpenMP in the laboratory. After the initial assignments solved in the laboratory, you have to complete and parallelise an application. The application you will work with is a simplification of a real application of the field of machine learning.

The aim is to apply all the knowledge acquired in class and worked on in the laboratory sessions to develop a correct and as efficient as possible parallel application. Use all the techniques learned to parallelise the application, even though, in some parts of the code, the execution times are very small and their parallelization may not be very efficient.

The practical work is designed to be carried out in a collaborative manner in groups of two people. Before starting to program, it is important to read this document carefully and analyse the code of the application (data structures, main program, objective of the functions to be programmed ....).

Genetic analysis of samples of pathogenic elements

In view of the situation that has arisen with the COVID-19 and to better cope with new pandemics, the WHO wants to carry out a genetic analysis of a set of samples available in its laboratories. The WHO has a data bank of around 200,000 genetic samples of pathogenic elements. Each of them is identified with 40 genetic characteristics and the WHO knows what type of disease could produce such a sample. To this end, taking into account 20 families of possible diseases, it has catalogued the types of diseases that each of the laboratory samples may cause.

As part of a research project, you have been commissioned to process this database to classify each sample into 100 different genetic groups. The sample is placed in a genetic group according to the closeness of its genetic characteristics, integrating into one group those samples with the highest similarities or smallest distances.

To classify the samples into different genetic groups we will use **K-means** (Lloyd 1982) clustering algorithm, which is one of the most widely used algorithms in the literature to solve similar problems (classify examples into groups based on a set of characteristics). Its working principle consists of minimising the sum of square errors (see algorithm) among the elements of the same group. Next algorithm summarises the process.

```
Begin
  Training data                      $X = \{x_i,\ 1 \le i \le N\}$
  Select the number of clusters:     $K \le N$
  Randomly select K centroids:       $C = \{c_j,\ 1 \le j \le K\};$
  Repeat
    Assign the instances to the closest cluster centroids:
      for i in 1 to N
        closest $c(x_i) = c_i \mid \min_{1 \le j \le K} d(x_i, c_j)$
      end for;
    Update the K cluster centroids C:
      for j in 1 to K
        $c_j$ = mean (xi | closest $c(x_i) = j$);
      end for;
  Until cluster centroids stop changing or maximum number of iterations
End
```

The data file owned by the WHO contains information about more than 200000 instances with 40 genetic characteristics (normalised between 0 and 100) for each of them. **The aim is to group all the instances into 100 genetic groups** (number of clusters *K*=100) **according to the closeness of their 40 genetic characteristics**.

After the generation of the 100 clusters, you will be required to obtain some results. On the one hand, a **compactness value** is calculated (average distance between all the elements in a group) which indicates the degree of dispersion of the elements **in each cluster**.

On the other hand, an **analysis of the presence of the 20 diseases** in the samples **in each group** is made, obtaining the highest and lowest percentage of presence of each disease among all the groups and the groups that have these values (maximum and minimum).

The algorithm is designed in the following way:

Phase 1.

- At the beginning, two files are read. On the one hand, the database (`dbgen.dat`), which is stored in matrix `elem`. Each row of the matrix has the genetic information of a sample (40 values in 40 columns). On the other hand, the relationship of each sample with the 20 families of diseases it could produce (file `dbdise.dat`), which is stored in matrix `dise` (each row has 20 values, values between 0 and 1 indicating the probability of the sample to generate or not that type of disease).
- Then, 100 representatives or centroids are generated randomly, one for each of the genetic groups, in the 40 dimension space. Values for the 40 genetic characteristics are generated randomly for each of the 100 representatives.
- To find the closest cluster, the genetic distances between all the individuals and the 100 centroids need to be calculated using function `gendist(...)`. The genetic distance is the Euclidean distance. The Euclidean distance between points p and q is the length of the line segment connecting them. In Cartesian coordinates, if p = (p_1, p_2,..., p_n) and q = (q_1, q_2,..., q_n) are two points in Euclidean n-space, then the distance (d) from p to q, or from q to p is given by the Pythagorean formula:

  $$d(p,q) = \text{square root } [(p_1-q_1)^2 + (p_2-q_2)^2 +\ldots+.(p_n-q_n)^2]$$

  Each element will be assigned to the group with the closest centroid in function `closestgroup(...)`.
- Once every element has been assigned a group, centroids need to be updated. The new centroids are calculated for each group by averaging all the element of the group in their 40 dimensions.
- Distances between old centroids and new centroids are calculated.
- This procedure is iterated until practically the same instances are assigned to each cluster, that is, the movement of the centroids practically disappears (is smaller than `DELTA`, a threshold value), or a particular number of iterations, `MAXIT`, is achieved.

Phase 2.

- After the groups have been established the compactness of each group needs to be calculated. To obtain the compactness of each group, the average distance between all the individuals of the group must be calculated in function `compactness(...)`.

  In addition, the analysis of the presence of each disease in the groups is carried out. Function `diseases(...)` is used with this aim.

All the material you need to complete and parallelise the application is as usual in **`ARC/Genetics`** directory in your account. Copy the files to one of your directories. **NOTE: do not copy the input files (`dbgen.dat and dbdise.dat);` they are too big, so we will use directly the ones in the `ARC/Genetics` directory**

▪ `gengroups_s.c`	Main program of the application (serial version).
▪ `fungg_s.c`	It contains the functions used in the main program: `gendist, closestgroup, compactness` and `diseases`. You have to complete them to generate the serial version.
▪ `definegg.h`	Definition of the constants used in files `gengroups_s.c` and `fungg_s.c`
▪ `fungg.h`	Function declarations as `extern`.
▪ `dbgen.dat`	Input file containing the genetic characteristics of all the elements. The first line in the file indicates the number of elements contained. The rest of the lines contain the 40 genetic characteristics for each instance.
▪ `dbdise.dat`	Input file containing 20 values between 0 and 1 (20 diseases) for each sample These values indicate the probability of the sample to generate or not that type of disease.
▪ `results.out`	File containing the results to be obtained: centroids and group density, and disease analysis. In this way, you can compare the results you obtain with those you should get.

To compile and execute the program:

- To compile the complete program:

  ```
  gcc -O2 -o gengroups_s gengroups_s.c fungg_s.c -lm
  ```

- To execute the program (assuming you have the material in a directory created from the root directory of your account) [you can create a command to launch the execution of the program]:

  ```
  gengroups_s../ARC/Genetics/dbgen.dat ../ARC/Genetics/dbdise.dat [num]
  ```

  There are two options for execution. If the third parameter —`num`— is indicated, only the indicated number of elements is processed; you have to use this option with small numbers for fast executions. If the third parameter is not indicated, the complete input file is processed; that is what you have to do at the end, after you check that the program works correctly, to obtain results.

To carry out this practical work, follow the next steps. Remember that you will work in groups, but each group's work is individual.

A. Generate the serial version of the code

A1. Analyse the structure of the serial program and complete the incomplete parts (in file `fungg_s.c`).

A2. Generate a command or script to compile all the modules of the application. Edit a text file with the command (ex., `gcc -O2 -o gengroups_s gengroups_s.c fungg_s.c -lm`) and then change permissions using `chmod 700 command_file` to make it executable. This way, it will be enough to execute that command every time you want to compile the application, instead of having to type every time the complete (long) command in the terminal command line. Remember that you need to include `-lm` option to compile some mathematical functions.

A3. **Prove that the serial program works correctly.** Compare the results you obtain in file `results_s.out` with those in file `results.out`.

To check that the results are the same you can always compare the output in the screen or compare two files. For instance:

```
diff results.out  results_s.out
```

To make the initial tests, you also have a reduced results file, for 1000 samples `(results1000.out)`.

B. Implement the parallel version of the program

B1. Once the serial version is correct, implement the parallel version of the program using OpenMP. Create a copy of all modules of the serial version and modify it to program the parallel version (e.g. `gengroups_p.c`, and so on for the other modules). In this way, you will always have the complete serial version of the program available.

Create a new script to compile the parallel version of the application.

Prove that it works correctly in two stages: (a) the results you obtain with the serial and the parallel version (with 3 threads for instance) are the same; and (b) the results you obtain with different numbers of threads (ex. with 2 and 7) are the same.

B2. **Analyse the performance you obtain** depending on the number of threads and try to optimise the parallel code. For instance, take into account the different scheduling strategies, and the need of synchronisation functions to propose the most adequate solution.

Select the best option and execute the parallel version of the program for instance with **2, 4, 8, 16, 24 and 32** processors. Obtain serial and parallel execution times and calculate speed-ups and efficiency.

Obviously, do all the tests and experiments that you consider appropriate.

C. Write a technical report

Finally you need to write a technical report that includes a clear explanation of the proposed programs implementations (serial and parallel), the obtained results, data and the corresponding figures, etc. **The report must reflect all the work done and must be done according to the piece of advice given for reports.**

>> **Stimated working times** (groups of two people): 40 hours

- Analyse the application, understand it and generate the serial version:	8 - 10 hours
- Create the parallel version, checkings, trials, results:	18 - 22 hours
- Writting a technical report:	8 - 10 hours

>> **Delivery date** (ireport and code in eGela): 30th December

```
/*
    CA - practical work OpenMP
    gengroups_s.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_s.c and include option -lm
*****************************************************************/

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

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

float        elem[MAXELE][NFEAT];       // matrix to keep info about every element
struct ginfo iingrs[NGROUPS];           // vector to store info about each group

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


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

void main (int argc, char *argv[])
{
  float   cent[NGROUPS][NFEAT], newcent[NGROUPS][NFEAT]; //centroid and new centroid
  double  additions[NGROUPS][NFEAT+1];
  float   compact[NGROUPS];                          // compactness of each cluster

  int     i, j;
  int     nelem, group;
  int     grind[MAXELE];   //group assigned to each element
  int     finish = 0, niter = 0;
  float   discent;

  FILE    *f1, *f2;
  struct timespec  t1, t2, t3, t4, t5, t6, t7;
  double  t_read, t_clus, t_org, t_compact, t_anal, t_write;


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

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


  // read data from files: elem[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", &nelem);
  if (argc == 4) nelem = atoi(argv[3]);
  for (i=0; i<nelem; i++)
  for (j=0; j<NFEAT; j++)
    fscanf (f1, "%f", &(elem[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<nelem; i++)
  for (j=0; j<TDISEASE; j++)
    fscanf (f1, "%f", &dise[i][j]);
  fclose (f1);

  clock_gettime (CLOCK_REALTIME, &t2);


  // select randomly the first centroids
  // ===================================

  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];
  }


  // Phase 1: classify elements and calculate new centroids
  // ======================================================

  niter = 0; finish = 0;
  while ((finish == 0) && (niter < MAXIT))
  {
    // Obtain the closest group or cluster for each element
    closestgroup (nelem, elem, cent, grind);

    // 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<nelem; i++)
    {
      for (j=0; j<NFEAT; j++)
        additions[grind[i]][j] += elem[i][j];
      additions[grind[i]][NFEAT] ++;
    }

    // Calculate new centroids and decide to finish or not depending on DELTA
    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 = gendist (&newcent[i][0], &cent[i][0]);
        if (discent > DELTA) finish = 0 // change in one of the dimensions; continue

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

  clock_gettime (CLOCK_REALTIME, &t3);
```

```
// Phase 2: calculate the "compactness" of the group
//          and analyse diseases
//==================================================

for (i=0; i<NGROUPS; i++) iingrs[i].size = 0;

// number of elements and classification
for (i=0; i<nelem; i++)
{
  group = grind[i];
  iingrs[group].members[iingrs[group].size] = i;
  iingrs[group].size ++;
}

clock_gettime (CLOCK_REALTIME, &t4);


// compactness of each group: average distance between elements
compactness (elem, iingrs, compact);

clock_gettime (CLOCK_REALTIME, &t5);


// diseases analysis
diseases (iingrs, dise, disepro);

clock_gettime (CLOCK_REALTIME, &t6);


// write results in a file
// =======================

f2 = fopen ("results_s.out", "w");
if (f2 == NULL) {
  printf ("Error when opening file %s \n", argv[1]);
  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", newcent[i][j]);
  fprintf (f2,"\n");
}
fprintf (f2, "\n\n Number of elements and compactness of the grousp\n\n");
for (i=0; i<NGROUPS; i++)
  fprintf (f2, " %6d  %.3f \n", iingrs[i].size, compact[i]);
fprintf (f2, "\n\n Analisis of diseases\n\n");
fprintf (f2,"\n Dise.   Max - Clus    Min - Clus");
fprintf (f2,"\n ================================\n");
for (i=0; i<TDISEASE; i++) {
  fprintf (f2,"  %2d     %4.2f - %2d     %4.2f - %2d", i, disepro[i].max,
                disepro[i].gmax, disepro[i].min, disepro[i].gmin);
  fprintf (f2,"\n");
}
fclose (f2);

clock_gettime (CLOCK_REALTIME, &t7);

// print some results
// ==================

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_org     = (t4.tv_sec-t3.tv_sec) + (t4.tv_nsec-t3.tv_nsec) / (double)1e9;
t_compact = (t5.tv_sec-t4.tv_sec) + (t5.tv_nsec-t4.tv_nsec) / (double)1e9;
t_anal    = (t6.tv_sec-t5.tv_sec) + (t6.tv_nsec-t5.tv_nsec) / (double)1e9;
t_write   = (t7.tv_sec-t6.tv_sec) + (t7.tv_nsec-t6.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_org:     %6.3f s", t_org);
  printf ("\n    T_compact: %6.3f s", t_compact);
  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_org + t_compact + t_anal
                     + t_write);


  printf ("\n centroids 0, 40 and 80 and the compactness of their group\n ");
  for (i=0; i<NGROUPS; i+=40) {
    printf ("\n  z%2d -- ", i);
    for (j=0; j<NFEAT; j++) printf ("%5.1f", cent[i][j]);
    printf ("\n          %5.6f\n", compact[i]);
  }

  printf ("\n >> Sizes of the groups \n");
  for (i=0; i<10; i++) {
    for (j=0; j<10; j++) printf ("%7d", iingrs[10*i+j].size);
    printf("\n");
  }

  printf ("\n >> DISEASES \n");
  printf ("\n Dise.   Max - Clus    Min - Clus");
  printf ("\n ================================\n");
  for (i=0; i<TDISEASE; i++) {
    printf ("  %2d     %4.2f - %2d     %4.2f - %2d", i, disepro[i].max,
                     disepro[i].gmax, disepro[i].min, disepro[i].gmin);
    printf("\n");
  }
}
```

```
/*
   CA - OpenMP
   fungg_s.c
   Routines used in gengroups_s.c program
   TO BE COMPLETED
****************************************************************/

#include <math.h>
#include <float.h>
#include "definegg.h"            // definition of constants and structs



/* 1 - Function to calculate the genetic distance; Euclidean distance between two elements.
       Input:   two elements of NFEAT characteristics (by reference)
       Output:  distance (double)
*********************************************************************************************/

double gendist (float *elem1, float *elem2)
{
   // TO DO
   // calculate the distance between two elements (Euclidean)
}



/* 2 - Function to calculate the closest group (closest centroid) for each element.
   Input:  nelem  number of elements, int
           elem   matrix, with the information of the elements, of size MAXELE x NFEAT, by ref
           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 nelem, float elem[][NFEAT], float cent[][NFEAT], int *grind)
{
   // TO DO
   // grind: closest group/centroid for each element
}



/* 3 - Function to calculate compactness of each group (avg dist between all elem in a group)
   Input:  elem     elements (matrix of size MAXELE x NFEAT, by reference)
           iingrs   indices of the elems in each group (matrix of size NGROUPS x MAXELE, by ref)
   Output: compact  compactness of each group (vector of size NGROUPS, by reference)
                    if only 0 or 1 elements, compact = 0
**********************************************************************************************/

void compactness (float elem[][NFEAT], struct ginfo *iingrs, float *compact)
{
   // TO DO
   // compactness of each group: average distance between members
}



/* 4 - Function to analyse diseases
   Input:  iingrs   indices of the elems in each group (matrix of size NGROUPS x MAXELE, by ref)
           dise     information about the diseases (NGROUPS x TDISEASE)
   Output: disepro  analysis of the diseases: maximum, minimum, and groups
*********************************************************************************************/

void diseases (struct ginfo *iingrs, float dise[][TDISEASE], struct analysis *disepro)
{
   // TO DO
   // process the information about diseases to obtain
   // the maximum and the group where the maximum is found (for each disease)
   // the minimum and the group where the minimum is found (for each disease)
}
```

```
/*
   definegg.h
   Constants used in files gengrops_s.c and fungg_s.c
***************************************************************/

#define MAXELE   230000   // number of elements (samples)
#define NGROUPS  100      // number of clusters
#define NFEAT    40       // features of each instance
#define TDISEASE 20       // types of disease

#define DELTA    0.01     // convergence: minimum change in centroids
#define MAXIT    1000     // convergence: maximum number of iterations


struct ginfo              // information about groups
{
 int  members[MAXELE];    // members
 int  size;               // number of elements
};


struct analysis           // analysis of diseases
{
 float  max, min;         // maximum and minimum for each disease
 int    gmax, gmin;       // groups for maximums and minimums
};
```

```
/*
   fungg.h
   headers of the functions used in gengroups_s.c
***************************************************************/

extern double gendist
       (float *elem1, float *elem2);

extern void   closestgroup
       (int nelem, float elem[][NFEAT], float cent[][NFEAT], int *grind);

extern void   compactness
       (float elem[][NFEAT], struct ginfo *iingrs, float *compact);

extern void   diseases
       (struct ginfo *iingrs, float dise[][TDISEASE], struct analysis *disepro);
```