Facultad de Informática
UPV/EHU

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea

Grado en Ingeniería Informática

Arquitectura de Computadores

GII – 2º curso

Sistemas multiprocesador

Trabajo práctico: paralelización de una aplicación utilizando **OpenMP**

En el tercer tema de la asignatura estamos analizando las características principales de los sistemas multiprocesador y, para programar aplicaciones paralelas en estos sistemas, estamos utilizando OpenMP. Tras los ejercicios iniciales realizados en el laboratorio, tienes que completar y paralelizar una aplicación. Se trata de una simplificación de una aplicación real, del ámbito del aprendizaje automático.

El objetivo es aplicar todos los conocimientos vistos en clase y trabajados en los laboratorios para desarrollar una aplicación paralela correcta y lo más eficiente posible. Utiliza todas las técnicas aprendidas para paralelizar la aplicación, a pesar de que, en algunas partes del código, los tiempos de ejecución son muy pequeños y su paralelización quizá no sea muy eficiente.

El trabajo está pensado para que se realice en grupos de dos personas, de forma colaborativa. Antes de empezar a programar, es importante leer con atención esta documentación y analizar el código de la aplicación (estructuras de datos, programa principal, objetivo de las funciones que se deben programar....).

Análisis genético de muestras de elementos patógenos

A la vista de la situación que se ha dado con el COVID, para enfrentarse mejor ante nuevas pandemias, la OMS (Organización Mundial de la Salud) quiere hacer un análisis genético de un conjunto de muestras de las que dispone en sus laboratorios. La OMS dispone de un banco de datos con alrededor de 211.000 muestras genéticas de elementos patógenos (**`dbgen.dat`**). Cada una de ellas está identificada con 40 características genéticas y la OMS sabe qué tipo de enfermedad podría llegar a producir dicha muestra. Para ello, teniendo en cuenta 18 familias de posibles enfermedades, ha catalogado los tipos de enfermedades que puede ocasionar cada una de las muestras de laboratorio (**`dbenf.dat`**).

Dentro de un proyecto de investigación, te han encargado procesar ese banco de datos, para encuadrar cada muestra en su correspondiente grupo genético. La muestra se encuadra en un grupo genético en función de la cercanía de sus características genéticas, integrando en un grupo aquellas muestras con las mayores similitudes.

**** K-means ****

Para clasificar las muestras en su grupo genético se utilizará el algoritmo de *clustering* **K-means** (Lloyd 1982). La técnica de clustering o agrupamiento es un método no-supervisado de clasificación de patrones, el cual particiona el espacio de entrada en diferentes clústeres. El objetivo de los algoritmos de clustering es crear particiones de clústeres de tal manera que los ejemplos que pertenecen al mismo clúster sean similares y los ejemplos que pertenecen a clústeres distintos, diferentes. K-means es uno de los algoritmos más utilizados en la literatura científica y su principio principal es minimizar las la suma de errores al cuadrado (véase el algoritmo) entre los ejemplos del mismo grupo. El algoritmo es el siguiente:

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 ($X_i \mid$ closest $C(X_i) = j$);
      end for;
  Until cluster centroids stop changing or maximum number of iterations
End

Cada agrupación o clúster se representa por el punto céntrico que se calcula haciendo la media de todos los elementos del clúster. Este punto céntrico es conocido como “centroide”.

El algoritmo K-means genera particiones de, como mucho, K clústeres/ agrupaciones, pero K es un hiper-parámetro que debe de ser dado al algoritmo, y el usuario no sabe cual es el número de clústeres idóneo. Por tanto, una vez que el algoritmo de clustering haya procesado la base de datos y obtenido una partición de clústeres que particiona los datos de entrada en K grupos, emerge una cuestión relevante: ¿Cómo de bien se ajusta la partición a los datos de entrada? ¿Ha sido el K seleccionado una buena elección?

Para ello, la metodología que se sigue en un proceso de clustering es el siguiente: se computan diferentes particiones de clústeres y se selecciona la partición que mejor se ajusta a los datos de entrada. Como esta información (K) no es previamente conocida, tendremos que ejecutar el algoritmo varias veces con diferentes valores de K. En nuestro caso, el K inicial lo fijaremos a

35 y lo incrementaremos por 10 en cada iteración, obteniendo particiones de 35, 45, 55... clústeres. En este proceso evaluaremos todas las particiones y seleccionaremos una de las mejores en ajustarse a los datos de entrada.

El proceso de estimar cómo de bien se ajusta una partición de datos a la estructura que subyace en los datos se conoce como la validación de clústeres (Cluster Validation). Las técnicas más comunes validan una partición utilizando los mismos datos particionados. Para ello miden o aproximan la compactitud intra-clúster y la separación inter-clúster. Estos indicadores se llaman Índices de Validación de Clústeres (Cluster Validity Indexes / CVI). En este proyecto vamos a utilizar uno de estos índices para validar y seleccionar las particiones que generemos.

**** Índices de Validación de Clústeres (Cluster Validity Indexes / CVI) ****

Nuestro CVI, por un lado, calculará la compactitud de cada clúster o grupo (cuan compactos están los elementos del clúster) como el promedio de las distancias entre pares de los elementos del grupo (término a). Por otro lado, la separación de un clúster se calculará como el promedio de las distancias entre el centroide del clúster en cuestión y los otros centroides (término b). A continuación se describirá el índice que se va ha utilizar:

- **El término a(k) aproxima la distancia intra-clúster del clúster k**, y para ello calcula la media de todas las distancias entre elementos que pertenecen al clúster k. Nos da la idea de cuan cohesionados/ compactos están los elementos dentro del clúster k o qué densidad tiene el clúster k. Por el bien de la simplicidad y la comprensión esta fórmula está sin optimizar.

$$a(k) = \begin{cases} \frac{1}{|c_k|^2} \sum_{x_i \in c_k} \sum_{x_j \in c_k} d(x_i, x_j) & |c_k| > 1, \\ 0 & |c_k| \leq 1. \end{cases}$$

  donde c_k es el centroide del clúster k, x_i y x_j son los elementos i y j que están asociados al centroide c_k o al clúster k, d() es la distancia euclidiana entre dos ejemplos y $|c_k|$ es el número de ejemplos que pertenecen al clúster k.

- **El término b(k) aproxima la distancia inter-clúster del clúster k**, y para ello calcula la media de todas las distancias entre el centroide k y los otros centroides. Nos da la idea de cuan separado está el clúster k de otras.

$$b(k) = \frac{1}{|C| - 1} \sum_{c_p \in C} d(c_k, c_p)$$

  donde C es la colección de centroides, |C| el número de centroides o el número de clústeres que tiene la partición de clústeres y c_k y c_p son los centroides k y p.

- **El término s(k) es un ratio entre el termino a(k) y b(k)** que indica la calidad del clúster k. En cuanto a este termino, lo deseable, es que la distancia intra-clúster (término a) sea pequeña (cohesión) y la distancia inter-clúster (término b) grande (separación).

$$s(k) = \frac{b(k) - a(k)}{max\{a(k), b(k)\}}$$

- **S es la media de todos los términos s(k)** e indica la calidad de la partición de clústeres.

$$S = \frac{1}{|C|} \sum_{c_k \in C} s(k)$$

Definido así, el rango de este índice es [-1, 1], donde -1 indica una mala partición y el 1 una buena partición.

**** Análisis de enfermedades ****

Como se ha comentado, el banco de datos contiene 40 datos genéticos (valores normalizados entre 0 y 100) de más de 211.000 muestras. El objetivo es agrupar todas esas muestras en K poblaciones (*clústeres*) diferentes, en función de la cercanía entre sus 40 características genéticas.

Tras la generación de los K clústeres, se realizará un análisis de la presencia de las 18 enfermedades (**`dbenf.dat`**) en las muestras encuadradas en cada grupo, obteniendo cuál es el mayor y el menor porcentaje de presencia de cada enfermedad entre todos los grupos y los grupos que tienen esos valores (máximo y mínimo). La presencia de una enfermedad en un grupo se calculará como la mediana de las probabilidades correspondientes a esa enfermedad en el grupo en cuestión.

Se utilizará una versión simplificada de la mediana: se ordenarán los valores del conjunto de menor a mayor y se devolverá el valor que está en la posición N/2 (división entre enteros) tanto si N es impar como par.

**** Fases de la aplicación ****

La aplicación funciona de esta forma:

1ª fase

1. Al comienzo se leen dos ficheros. Por una parte, el banco de muestras (`dbgen.dat`), que se almacena en la matriz `elem`. Cada fila de la matriz tiene la información genética de una muestra (40 valores en 40 columnas). Por otra parte, la vinculación de dicha muestra con las 18 familias de enfermedades que podría llegar a producir (fichero `dbenf.dat`), que se almacena en la matriz `enf` (cada fila tiene 18 valores, valores entre 0 y 1 que indican la probabilidad de que la muestra genere o no ese tipo de enfermedad).
2. A continuación empieza un proceso iterativo (desde el paso 3 a 8) en el que se generá la partición y se evalúa.
3. Se genera de forma aleatoria un *centroide* inicial para cada uno de los K grupos (clústeres). En este caso como el K inicial es 35 se generarán 35 centroides en un espacio de 40 dimensiones que corresponden a las características genéticas de estos centroides o representantes del clúster.
4. Partiendo de esa situación, se calcula la distancia genética entre todos los elementos del fichero y los 100 *centroides*, utilizando la función `gendist`. La distancia genética entre dos elementos es simplemente la distancia euclídea entre los 40 valores de sus características genéticas.

   $$\text{dis}(p, q) = \text{raiz\_cuadrada}\ [\ (p_1 - q_1)^2 + (p_2 - q_2)^2 + \ldots + (p_n - q_n)^2\ ]$$

   Las distancias calculadas se utilizan para hacer los grupos: se asigna a cada elemento del fichero su grupo genético más cercano, el de menor distancia, mediante la función `grupo_cercano (…)`

5. Una vez clasificados los elementos en K grupos, se calcula un nuevo *centroide* para cada grupo, esto es, se calculan los valores medios para las 40 características de todos los elementos de ese grupo.
6. Se calcula la distancia entre el nuevo *centroide* y el *centroide* anterior.
7. Se repite el proceso hasta que se cumpla una de estas dos condiciones: (a) el movimiento de los valores de los *centroides* es inferior a un determinado umbral (DELTA1), (b) se supera un número máximo de iteraciones (MAXIT).
8. Una vez que la partición está generado será evaluado utilizando el CVI. Para ello, se comparará la bondad de la partición (valor del CVI) actual con la de la anterior y si la bondad actual es mejor el número de clústeres (K) se incrementará en 10 y el proceso se repetirá. Si la calidad o bondad de la partición actual (según el CVI) no mejora la de la partición anterior, la búsqueda del K se para y se usa esta última partición en el fase siguiente. Esta estrategia de búsqueda es conocida como “best-first”.

2ª fase

1. Una vez finalizado el proceso de clusterización, se realiza el análisis de presencia de cada enfermedad en los grupos. Analizando cada grupo hay que calcular en cual de los grupos hay la probabilidad máxima y mínima de que exista una enfermedad, utilizando como la probabilidad representativa del grupo la mediana de cada enfermedad en ese grupo. Para ello, se utiliza la función `analisis_enfermedades` **`(...).`**

   Tened en cuenta que para calcular la mediana hay que ordenar las probabilidades de una enfermedad del grupo en cuestión y hay que quedarse con el valor medio (N/2 tanto para N impar como par). Ordena las probabilidades utilizando un algoritmo simple de ordenar, como el de la burbuja.

**** Material proporcionado ****

Tienes todo el material necesario para desarrollar la aplicación en el directorio **ARQ/pgenetica** de tu cuenta de trabajo. Copia el material a un nuevo directorio de trabajo en tu cuenta. **NOTA: no copies los ficheros de entrada (`dbgen.dat` y `dbenf.dat`); son demasiados grandes, por lo que utilizaremos directamente los que están en el directorio ARQ/pgenetica**

- `gengrupos_s.c` Programa principal de la aplicación (versión serie).
- `fun_s.c` Contiene las funciones que utiliza el programa principal: `gendist`, `grupo_cercano`, `silhouette_simple` y `analizar_enfermedades`.

  Tienes que completar esas funciones para conseguir la versión **serie** del programa.
- `defineg.h` Definiciones que se utilizan en los ficheros `gengrupos_s.c` y `fun_s.c`.
- `fun.h` Cabeceras (declaraciones) de las funciones (`extern`).
- `dbgen.dat` Fichero de entrada que contiene los datos genéticos de todas las muestras. El valor de la primera línea indica el número de muestras y los valores del resto de líneas (40 valores) son las características genéticas de cada muestra.
- `dbenf.dat` Fichero de entrada que contiene 18 valores entre 0 y 1 (18 enfermedades) por cada muestra. Estos valores indican la probabilidad de que la muestra genere o no ese tipo de enfermedad.
- `res.out` Fichero que contiene los resultados que debes conseguir: centroides y densidad de los grupos, y análisis de enfermedades. De esta forma, puedes comparar tus resultados (en el fichero `dbgen_s.out` para la versión serie del programa) con los que deberías obtener.

**** Para compilar y ejecutar el programa ****

- Para **compilar** todo el programa:

  ```
  gcc -O2 -o gengrupos_s gengrupos_s.c fun_s.c -lm
  ```

- Para **ejecutar** el programa (suponiendo que tienes el material en un directorio creado desde el directorio raíz de tu cuenta) [puedes crear un comando para lanzar la ejecución del programa]:

  ```
  gengrupos_s ../ARQ/pgenetica/dbgen.dat ../ARQ/pgenetica/dbenf.dat [ num ]
  ```

  Hay dos opciones para ejecutar el programa. Si se indica el tercer parámetro —`num`—, se procesará sólo el número de muestras indicado. Es la opción que puedes utilizar para realizar pruebas, utilizando un número pequeño de muestras para que la ejecución sea rápida. Si no se indica el tercer parámetro, se procesa todo el fichero de entrada. Esta opción es la que utilizarás, tras comprobar que el programa funciona correctamente, para obtener los resultados finales.

RECUERDA que trabajaremos en grupo, pero el trabajo de cada grupo es individual.

Para realizar este trabajo práctico sigue los siguientes pasos.

A. Crea la versión serie de la aplicación

A1. Analiza la estructura del programa en serie y completa el código (en el fichero `fun_s.c`).

A2. Crea un comando (*script*) para compilar toda la aplicación. Para ello, edita un fichero de texto con los comandos que quieres ejecutar (por ejemplo, `gcc -O2 -o gengrupos_s gengrupos_s.c fun_s.c -lm`). Cambia los permisos del fichero para convertirlo en un fichero ejecutable: `chmod 700 nombre_fichero`

De esta forma, será suficiente ejecutar ese comando cada vez que quieras compilar la aplicación, en lugar de tener que escribir cada vez el comando completo (largo) en la línea de comandos de la terminal. No olvides que debes finalizar el comando de compilación con la opción `-lm`, para poder incorporar la librería que incluye las funciones matemáticas que utilizamos.

A3. **Comprueba el correcto funcionamiento de la versión serie del programa**. Compara tus resultados (en el fichero `dbgen_s.out`) con los resultados del fichero `res.out`. Para ello, puedes visualizar algunas líneas de ambos ficheros o comparar ambos ficheros utilizando el siguiente comando Linux:

```
diff res.out dbgen_s.out
```

Para hacer las pruebas iniciales, dispones también de un fichero de resultados reducido, para 1000 muestras (`res1000.out`).

B. Crea la versión paralela de la aplicación

B1. Tras crear la versión serie y comprobar su correcto funcionamiento, crea la versión paralela de la aplicación utilizando OpenMP. Crea una copia de todos los módulos de la versión serie y modifícala para programar la versión paralela (por ejemplo, `gengrupos_p.c`, y así para el resto de módulos). De esta forma, siempre tendrás disponible la versión serie completa del programa.

Crea otro script para poder compilar la versión paralela de la aplicación.

Comprueba el correcto funcionamiento de la versión paralela de estas dos formas: (a) en serie y paralelo (por ejemplo, utilizando 3 hilos) comprobando que los resultados son

idénticos; y (b) en paralelo con diferentes hilos (por ejemplo, con 2 y 8) para comprobar que los resultados también son idénticos independientemente del número de hilos que se utilicen.

B2. A continuación, **analiza el rendimiento obtenido** en función del número de hilos. Intenta optimizar el código paralelo (estrategias de planificación y funciones de sincronización) para obtener una solución eficiente.

Una vez obtenida la "mejor" versión paralela, realiza la experimentación correspondiente: ejecuta la aplicación para **2, 4, 8, 16, 24, 32, 48** y **64** hilos; calcula los tiempos de ejecución serie y paralelo para ese número de procesos, los factores de aceleración y eficiencias conseguidas.

Obviamente, realiza todas las pruebas y experimentos que te parezcan oportunos.

C. Escribe un informe técnico

Finalmente, tienes que escribir un informe técnico: programas desarrollados, correctamente comentados y explicados, resultados obtenidos (datos y gráficas realizadas), conclusiones, etc. El informe es reflejo de todo el trabajo realizado, tómatelo con tiempo y seriedad. Ten en cuenta las recomendaciones que te hemos indicado en el documento correspondiente (ver documento en eGela).

>> Tiempos de trabajo estimados (grupos de dos personas): 40 horas

- Analizar la aplicación, entenderla y crear la versión serie:	08 - 10 horas
- Crear la versión paralela, comprobaciones, pruebas, resultados:	18 - 22 horas
- Escritura del informe técnico:	08 - 10 horas

>> **Plazo de entrega** (informe y código en eGela): **30 de diciembre**

gengrupos_s.c

```
/*****************************************************************
    gengrupos_s.c   SERIE
    Entrada: dbgen.dat    fichero con la informacion genetica de cada muestra
             dbenf.dat    fichero con la informacion sobre las enfermedades de cada muestra
    Salida:  dbgen_s.out  centroides, densidad, analisis
    compilar con el modulo fun_s.c y la opcion -lm
*****************************************************************/

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

#include "defineg.h"
#include "fun.h"

float  elem[MAXE][NCAR];              // elementos (muestras) a procesar
struct lista_grupos listag[MAX_GRUPOS];   // lista de elementos de los grupos

float  enf[MAXE][TENF];               // enfermedades asociadas a las muestras
struct  analisis prob_enf[TENF];      // analisis de los tipos de enfermedades

int ngrupos = 35;

// programa principal
// ==================
int main (int argc, char *argv[]) {
        float   a[MAX_GRUPOS]; // densidad de cada cluster
        int     popul[MAXE]; // grupo de cada elemento
        float   cent[MAX_GRUPOS][NCAR]; // centroides
        int     i, j, nelem, grupo, num, ind;
        int     fin = 0, num_ite = 0;
        int     convergencia_cont;
        double  sil, sil_old, diff;

        FILE   *fd;
        struct timespec  t1, t2;
        struct timespec  t11, t12, t17, t20, t21;
        double texe, t_lec, t_clust, t_enf, t_escri;

        if ((argc < 3)  || (argc > 4)) {
                printf ("[!] ERROR:  gengrupos bd_muestras bd_enfermedades [num_elem]\n");
                exit (-1);
        }

        setbuf(stdout, NULL);
        printf ("\n*** Ejecucion en serie ***\n\n");
        clock_gettime (CLOCK_REALTIME, &t1);


        // lectura de datos (muestras): elem[i][j]
        // =======================================
        clock_gettime (CLOCK_REALTIME, &t11);
        fd = fopen (argv[1], "r");
        if (fd == NULL) {
                printf ("[!] Error al abrir el fichero %s\n", argv[1]);
                exit (-1);
```

```
	}

	fscanf (fd, "%d", &nelem);
	if (argc == 4) nelem = atoi(argv[3]);     // 4. parametro: numero de elementos

	for (i=0; i<nelem; i++)
		for (j=0; j<NCAR; j++)
			fscanf (fd, "%f", &(elem[i][j]));

	fclose (fd);

	// lectura de datos (enfermedades): enf[i][j]
	// ==========================================
	fd = fopen (argv[2], "r");
	if (fd == NULL) {
		printf ("[!] Error al abrir el fichero %s\n", argv[2]);
		exit (-1);
	}

	for (i=0; i<nelem; i++) {
		for (j=0; j<TENF; j++)
			fscanf (fd, "%f", &(enf[i][j]));
	}
	fclose (fd);
	clock_gettime (CLOCK_REALTIME, &t12);
	t_lec = (t12.tv_sec-t11.tv_sec) + (t12.tv_nsec-t11.tv_nsec)/(double)1e9;

	convergencia_cont = 0;
	sil_old = -1;
	while(ngrupos<MAX_GRUPOS && convergencia_cont<1){
		// generacion de los primeros centroides de forma aleatoria
		// ========================================================
		inicializar_centroides(cent);

		// A: agrupar los elementos en ngrupos clusteres
		// =============================================
		num_ite = 0;
		fin = 0;
		while ((fin == 0) && (num_ite < MAXIT)) {
			// calcular el grupo mas cercano
			grupo_cercano (nelem, elem, cent, popul);

			// calcular los nuevos centroides de los grupos
			fin = nuevos_centroides(elem, cent, popul, nelem);

			num_ite++;
		}


		// B. Calcular la "calidad" del agrupamiento
		// =========================================
		// lista de clusters: numero de elementos y su clasificacion
		for (i=0; i<ngrupos; i++) listag[i].nelemg = 0;
		for (i=0; i<nelem; i++){
			grupo = popul[i];
			num=listag[grupo].nelemg;
			listag[grupo].elemg[num] = i;  // elementos de cada grupo (cluster)
```

```
                        listag[grupo].nelemg++;
                }

                // silhouette simple: calidad de la particion
                sil = silhouette_simple(elem, listag, cent, a);

                // calcular la diferencia: estabilidad
                diff = sil - sil_old;
                if(diff < DELTA2) convergencia_cont++;
                else convergencia_cont = 0;
                sil_old = sil;

                ngrupos=ngrupos+10;
        }
        ngrupos=ngrupos-10;
        clock_gettime (CLOCK_REALTIME, &t17);
        t_clust = (t17.tv_sec-t12.tv_sec) + (t17.tv_nsec-t12.tv_nsec)/(double)1e9;

        // 2. fase: numero de elementos de cada grupo; analisis enfermedades
        // ================================================================
        // analisis de enfermedades
        clock_gettime (CLOCK_REALTIME, &t20);
        analisis_enfermedades (listag, enf, prob_enf);
        clock_gettime (CLOCK_REALTIME, &t21);
        t_enf = (t21.tv_sec-t20.tv_sec) + (t21.tv_nsec-t20.tv_nsec)/(double)1e9;

        // escritura de resultados en el fichero de salida
        // ===============================================
        fd = fopen ("dbgen_s.out", "w");
        if (fd == NULL) {
                printf ("[!] Error al abrir el fichero dbgen_out.s\n");
                exit (-1);
        }

        fprintf (fd,">> Centroides de los clusters\n\n");
        for (i=0; i<ngrupos; i++) {
                for (j=0; j<NCAR; j++) fprintf (fd, "%7.3f", cent[i][j]);
                fprintf (fd,"\n");
        }

        fprintf (fd,"\n\n>> Numero de clusteres: %d. Numero de elementos en cada cluster:\n\n", ngrupos);
        ind = 0;
        for (i=0; i<ngrupos/10; i++) {
                for (j=0; j<10; j++){
                        fprintf(fd, "%6d", listag[ind].nelemg);
                        ind++;
                }
                fprintf(fd, "\n");
        }
        for(i=ind; i<ngrupos; i++) fprintf(fd, "%6d", listag[i].nelemg);
        fprintf(fd, "\n");

        fprintf (fd, "\n>> Densidad de los clusters: b[i] \n\n");
        ind = 0;
        for (i=0; i<ngrupos/10; i++) {
                for (j=0; j<10; j++){
                        fprintf (fd, "%9.2f", a[ind]);
```

```
                ind++;
        }
        fprintf (fd, "\n");
}
for(i=ind; i<ngrupos; i++) fprintf(fd, "%9.2f", a[i]);
fprintf(fd, "\n");

fprintf (fd,"\n\n>> Analisis de enfermedades (medianas) en los grupos\n\n");
for (i=0; i<TENF; i++)
        fprintf (fd,"Enfermedad: %2d - mmax: %4.2f (grupo %2d) - mmin: %4.2f (grupo %2d)\n",
                i, prob_enf[i].mmax, prob_enf[i].gmax, prob_enf[i].mmin, prob_enf[i].gmin);

fprintf (fd,"\n\n");
fclose (fd);

// mostrar por pantalla algunos resultados
// =======================================
printf ("\n>> Centroides 0, 20, 40...\n");
for (i=0; i<ngrupos; i+=20) {
        printf ("\n  cent%2d -- ", i);
        for (j=0; j<NCAR; j++) printf ("%5.1f", cent[i][j]);
        printf("\n");
}

printf ("\n>> Numero de clusteres: %d. Tamanno de los grupos:\n\n", ngrupos);
ind = 0;
for (i=0; i<ngrupos/10; i++) {
        for (j=0; j<10; j++){
                printf ("%6d", listag[ind].nelemg);
                ind++;
        }
        printf ("\n");
}
for(i=ind; i<ngrupos; i++) printf("%6d", listag[i].nelemg);
printf("\n");

printf("\n >> Densidad de los clusters: b[i]\n\n");
ind = 0;
for (i=0; i<ngrupos/10; i++) {
        for (j=0; j<10; j++){
                printf("%9.2f", a[ind]);
                ind++;
        }
        printf("\n");
}
for(i=ind; i<ngrupos; i++) printf("%9.2f", a[i]);
printf("\n");

printf ("\n>> Analisis de enfermedades en los grupos\n\n");
for (i=0; i<TENF; i++)
        printf ("Enfermedad: %2d - max: %4.2f (grupo %2d) - min: %4.2f (grupo %2d)\n",
                i, prob_enf[i].mmax, prob_enf[i].gmax, prob_enf[i].mmin, prob_enf[i].gmin);

clock_gettime (CLOCK_REALTIME, &t2);
t_escri = (t2.tv_sec-t21.tv_sec) + (t2.tv_nsec-t21.tv_nsec)/(double)1e9;
texe = (t2.tv_sec-t1.tv_sec) + (t2.tv_nsec-t1.tv_nsec)/(double)1e9;

printf("\n");
```

```
        printf("t_lec,%f\n", t_lec);
        printf("t_clust,%f\n", t_clust);
        printf("t_enf,%f\n", t_enf);
        printf("t_escri,%f\n", t_escri);
        printf("Texe,%f\n", texe);
}
```

fun_s.c

```
/*****************************************************************************
    AC - OpenMP -- SERIE
    fun_s.c
    rutinas que se utilizan en el modulo gengrupos_s.c
*****************************************************************************/
#include <math.h>
#include <float.h> // DBL_MAX
#include <stdlib.h>

#include "defineg.h"           // definiciones

/*****************************************************************************************
   1 - Funcion para calcular la distancia genetica entre dos elementos (distancia euclidea)
       Entrada:  2 elementos con NCAR caracteristicas (por referencia)
       Salida:  distancia (double)
*****************************************************************************************/
double gendist (float *elem1, float *elem2)
{
        // PARA COMPLETAR
        // calcular la distancia euclidea entre dos vectores
        return 0.0;
}

/*****************************************************************************************
   2 - Funcion para calcular el grupo (cluster) mas cercano (centroide mas cercano)
   Entrada:  nelem  numero de elementos, int
             elem   elementos, una matriz de tamanno MAXE x NCAR, por referencia
             cent   centroides, una matriz de tamanno NGRUPOS x NCAR, por referencia
   Salida:   popul  grupo mas cercano a cada elemento, vector de tamanno MAXE, por referencia
*****************************************************************************************/
void grupo_cercano (int nelem, float elem[][NCAR], float cent[][NCAR], int *popul)
{
        // PARA COMPLETAR
        // popul: grupo mas cercano a cada elemento
}

/*****************************************************************************************
   3 - Funcion para calcular la calidad de la particion de clusteres.
       Ratio entre a y b. El termino a corresponde a la distancia intra-cluster.
       El termino b corresponde a la distancia inter-cluster.
   Entrada:  elem     elementos, una matriz de tamanno MAXE x NCAR, por referencia
             listag   vector de NGRUPOS structs (informacion de grupos generados), por ref.
             cent     centroides, una matriz de tamanno NGRUPOS x NCAR, por referencia
   Salida:   valor del CVI (double): calidad/ bondad de la particion de clusters
*****************************************************************************************/
double silhouette_simple(float elem[][NCAR], struct lista_grupos *listag, float cent[][NCAR], float a[]){
    //float b[ngrupos];
```

```
    // PARA COMPLETAR

    // aproximar a[i] de cada cluster: calcular la densidad de los grupos
    //          media de las distancia entre todos los elementos del grupo
    //          si el numero de elementos del grupo es 0 o 1, densidad = 0
    // ...

    // aproximar b[i] de cada cluster
    // ...

    // calcular el ratio s[i] de cada cluster
    // ...

    // promedio y devolver
    // ...
    return 0.0;
}

/*****************************************************************************************
   4 - Funcion para relizar el analisis de enfermedades
   Entrada:  listag   vector de NGRUPOS structs (informacion de grupos generados), por ref.
             enf      enfermedades, una matriz de tamaño MAXE x TENF, por referencia
   Salida:   prob_enf vector de TENF structs (informacion del análisis realizado), por ref.
*****************************************************************************************/
void analisis_enfermedades(struct lista_grupos *listag, float enf[][TENF], struct analisis *prob_enf)
{
        // PARA COMPLETAR
        // Realizar el análisis de enfermedades en los grupos:
        //      mediana máxima y el grupo en el que se da este máximo (para cada enfermedad)
        //      mediana mínima y su grupo en el que se da este mínimo (para cada enfermedad)
}

/*****************************************************************************************************
   OTRAS FUNCIONES DE LA APLICACION
*****************************************************************************************************/

void inicializar_centroides(float cent[][NCAR]){
        int i, j;
        srand (147);
        for (i=0; i<ngrupos; i++)
                for (j=0; j<NCAR/2; j++){
                        cent[i][j] = (rand() % 10000) / 100.0;
                        cent[i][j+(NCAR/2)] = cent[i][j];
                }
}

int nuevos_centroides(float elem[][NCAR], float cent[][NCAR], int popul[], int nelem){
        int i, j, fin;
        double discent;
        double additions[ngrupos][NCAR+1];
        float newcent[ngrupos][NCAR];

        for (i=0; i<ngrupos; i++)
                for (j=0; j<NCAR+1; j++)
                        additions[i][j] = 0.0;

        // acumular los valores de cada caracteristica (100); numero de elementos al final
```

```
        for (i=0; i<nelem; i++){
                for (j=0; j<NCAR; j++) additions[popul[i]][j] += elem[i][j];
                additions[popul[i]][NCAR]++;
        }

        // calcular los nuevos centroides y decidir si el proceso ha finalizado o no (en funcion de DELTA)
        fin = 1;
        for (i=0; i<ngrupos; i++){
                if (additions[i][NCAR] > 0) { // ese grupo (cluster) no esta vacio
                        // media de cada caracteristica
                        for (j=0; j<NCAR; j++)
                                newcent[i][j] = (float)(additions[i][j] / additions[i][NCAR]);

                        // decidir si el proceso ha finalizado
                        discent = gendist (&newcent[i][0], &cent[i][0]);
                        if (discent > DELTA1)
                                fin = 0;  // en alguna centroide hay cambios; continuar

                        // copiar los nuevos centroides
                        for (j=0; j<NCAR; j++)
                                cent[i][j] = newcent[i][j];
                }
        }
        return fin;
}
```

defineg.h

```
/*********************************************************
    defineg.h
    definiciones utilizadas en los modulos de la aplicacion
*********************************************************/

#define MAXE     211640   // numero de elementos (muestras)
#define MAX_GRUPOS  100   // numero de clusters
#define NCAR     40       // dimensiones de cada muestra
#define TENF     18       // tipos de enfermedad

#define DELTA1    0.01    // convergencia: cambio minimo en un centroide
#define DELTA2    0.01    // convergencia: Best First, Silhouette
#define MAXIT    10000    // convergencia: numero de iteraciones maximo

extern int ngrupos;

// estructuras de datos
//
struct lista_grupos  // informacion de los clusters
{
 int elemg[MAXE];   // indices de los elementos
 int nelemg;        // numero de elementos
};

struct analisis     // resultados del analisis de enfermedades
{
 float mmax, mmin;   // maximo y minimo de las medianas de las probabilidad de cada enfermedad
 int gmax, gmin;    // grupos con los valores maximo y minimoi de las medianas
};
```

fun.h

```
/*************************************************************
    fun.h
    cabeceras de las funciones utilizadas en el modulo gengrupos
*************************************************************/

extern void inicializar_centroides(float cent[][NCAR]);
extern int nuevos_centroides(float elem[][NCAR], float cent[][NCAR], int popul[], int nelem);

extern double gendist (float *elem1, float *elem2);
extern void grupo_cercano (int nelem, float elem[][NCAR], float cent[][NCAR], int *popul);
extern double silhouette_simple(float elem[][NCAR], struct lista_grupos *listag, float cent[][NCAR], float a[]);
extern void analisis_enfermedades (struct lista_grupos *listag, float enf[][TENF], struct analisis *prob_enf);
```