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 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 200.000 muestras genéticas de elementos patógenos. 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 20 familias de posibles enfermedades, ha catalogado los tipos de enfermedades que puede ocasionar cada una de las muestras de laboratorio.

Dentro de un proyecto de investigación, te han encargado procesar ese banco de datos para encuadrar cada muestra en 100 grupos genéticos distintos. 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.

Para clasificar a las muestras en su grupo genético se utilizará el algoritmo de *clustering* **K-means** (Lloyd 1982). Uno de los usos habituales de ese algoritmo es la resolución de este tipo de problemas: clasificación de un elemento en un grupo o clúster en función de una serie de características concretas de dicho elemento. Para ello, el algoritmo de *clustering* trata de minimizar las distancias entre los elementos 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

Como se ha comentado, el banco de datos contiene 40 datos genéticos (valores normalizados entre 0 y 100) de más de 200.000 muestras. El objetivo es agrupar todas esas muestras en 100 poblaciones (*clústeres*) diferentes, en función de la cercanía entre sus 40 características genéticas.

Tras la generación de los 100 clústeres, se obtendrán unos resultados. Por una parte, se calcula un valor de densidad (distancia media entre todos los elementos de un grupo) que viene a indicar el grado de dispersión de los elementos de un clúster.

Por otra parte, se realiza un análisis de la presencia de las 20 enfermedades 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 aplicación funciona de esta forma:

1ª fase

- 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 20 familias de enfermedades que podría llegar a producir (fichero `dbenf.dat`), que se almacena en la matriz `enf` (cada fila tiene 20 valores, valores entre 0 y 1 que indican la probabilidad de que la muestra genere o no ese tipo de enfermedad).
- A continuación se genera de forma aleatoria un *centroide* inicial para cada uno de los 100 grupos (clústeres).
- Partiendo de esa situación, se calcula la distancia 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 20 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 (...)`
- Una vez clasificados los elementos en 100 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.
- Se calcula la distancia entre el nuevo *centroide* y el *centroide* anterior.
- 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, (b) se supera un número máximo de iteraciones.

2ª fase

- Una vez finalizado el proceso de clusterización, se calcula la densidad de cada clúster, esto es, la distancia media entre todos los elementos del clúster. Para ello, se utiliza la función `calcular_densidad` **`(...).`**
- Además, se realiza el análisis de presencia de cada enfermedad en los grupos. Para ello, se utiliza la función `analizar_enfermedades` **`(...).`**

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`, `calcular_densidad` 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 20 valores entre 0 y 1 (20 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.

Para realizar este trabajo práctico sigue los siguientes pasos. Recuerda que trabajaremos en grupo, pero el trabajo de cada grupo es individual.

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 y 32** 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: 8 - 10 horas
- Crear la versión paralela, comprobaciones, pruebas, resultados: 18 - 22 horas
- Escritura del informe técnico: 8 - 10 horas

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

```
/*
    AC - Trabajo práctico
    gengrupos_s.c     VERSIÓN SERIE

    Procesa la información genética de una población para generar grupos

    Entrada: dbgen.dat    información genética de cada muestra
             dbenf.dat    información sobre las enfermedades de cada muestra
    Salida:  dbgen_s.out  centroides, densidad y enfermedades

    compilar con el módulo fun_s.c y la opción -lm
*****************************************************************************/

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

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

float  elem[MAXE][NCAR];               // elementos (muestras) a procesar
float  enf[MAXE][TENF];                // enfermedades asociadas a las muestras
struct lista_grupos listag[NGRUPOS];   // lista de elementos de los grupos

// programa principal
// ====================

void main (int argc, char *argv[])
{
  float  cent[NGRUPOS][NCAR], newcent[NGRUPOS][NCAR]; // centroides
  float  densidad[NGRUPOS];             // densidad de cada cluster
  struct analisis prob_enf[TENF];       // análisis de los tipos de enfermedades

  int     popul[MAXE];                  // grupo de cada elemento
  double  additions[NGRUPOS][NCAR+1];

  int     i, j, nelem, grupo, num;
  int     fin = 0, num_ite = 0;
  double  discent;
  FILE    *fd;

  struct  timespec  t1, t2;
  double  texe;


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

  printf ("\n >> Ejecución serie\n");
  clock_gettime (CLOCK_REALTIME, &t1);

  // lectura de datos (muestras): elem[i][j]
  // =======================================

  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 parámetro: número 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);



// generación de los primeros centroides de forma aleatoria
  // ========================================================
  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];
   }


  // 1. fase: clasificar los elementos y calcular los nuevos centroides
  // ======================================================================

  num_ite = 0; fin = 0;
  while ((fin == 0) && (num_ite < MAXIT))
  {
    // calcular el grupo más cercano
    grupo_cercano (nelem, elem, cent, popul);


    // calcular los nuevos centroides de los grupos
    // media de cada característica
    // acumular los valores de cada característica (100); número de elementos al final
    for (i=0; i<NGRUPOS; i++)
    for (j=0; j<NCAR+1; j++)
      additions[i][j] = 0.0;

    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
  // decidir si el proceso ha finalizado o no (en función de DELTA)
    fin = 1;
    for (i=0; i<NGRUPOS; i++)
    {
      if (additions[i][NCAR] > 0) // ese grupo (cluster) no está vacío
      {
        for (j=0; j<NCAR; j++) newcent[i][j] = additions[i][j] / additions[i][NCAR];

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

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

```
// 2. fase: número de elementos de cada grupo; densidad; análisis de enfermedades
// ================================================================================
  for (i=0; i<NGRUPOS; i++) listag[i].nelemg = 0;

  // número de elementos y su clasificación
  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++;
  }

  // densidad de cada cluster: media de las distancias entre todos los elementos
  calcular_densidad (elem, listag, densidad);

  // análisis de enfermedades
  analizar_enfermedades (listag, enf, prob_enf);


// 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", newcent[i][j]);
    fprintf (fd,"\n");
  }

  fprintf (fd, "\n\nNumero de elementos de cada cluster y densidad del cluster\n\n");
  for (i=0; i<NGRUPOS; i++)
    fprintf (fd, " %6d  %.3f \n", listag[i].nelemg, densidad[i]);

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

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

// mostrar por pantalla algunos resultados
// =======================================
  printf ("\n>> Centroides 0, 40 y 80, y su valor de densidad\n ");
  for (i=0; i<NGRUPOS; i+=40) {
    printf ("\n  cent%2d -- ", i);
    for (j=0; j<NCAR; j++) printf ("%5.1f", cent[i][j]);
    printf ("\n          %5.6f\n", densidad[i]);
  }

  printf ("\n>> Tamano de los grupos \n");
  for (i=0; i<10; i++) {
    for (j=0; j<10; j++) printf ("%7d", listag[10*i+j].nelemg);
    printf("\n");
  }

  printf ("\n>> Analisis de enfermedades en los grupos\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].max, prob_enf[i].gmax, prob_enf[i].min, prob_enf[i].gmin);

  printf ("\n >> Número de iteraciones: %d", num_ite);
  printf ("\n >> Tex (serie): %1.3f s\n\n", texe);
}
```

```
/*
    AC - OpenMP -- SERIE
     fun_s.c
     rutinas que se utilizan en el módulo gengrupos_s.c
     PARA COMPLETAR
*****************************************************************************/

#include <math.h>
#include <float.h>
#include "defineg.h"           // definiciones


/* 1 - Función para calcular la distancia genética entre dos elementos (distancia euclídea)
       Entrada:  2 elementos con NCAR características (por referencia)
       Salida:  distancia (double)
*******************************************************************************/

double gendist (float *elem1, float *elem2)
{
   // PARA COMPLETAR
   // calcular la distancia euclídea entre dos vectores
}


/* 2 - Función para calcular el grupo (cluster) más cercano (centroide más cercano)
   Entrada:  nelem  número de elementos, int
             elem   elementos, una matriz de tamaño MAXE x NCAR, por referencia
             cent   centroides, una matriz de tamaño NGRUPOS x NCAR, por referencia
   Salida:   popul  grupo más cercano a cada elemento, vector de tamaño MAXE, por referencia
*******************************************************************************/

void grupo_cercano (int nelem, float elem[][NCAR], float cent[][NCAR], int *popul)
{
   // PARA COMPLETAR
   // popul: grupo más cercano a cada elemento
}


/* 3 - Función para calcular la densidad del grupo (dist. media entre todos sus elementos)
   Entrada:  elem     elementos, una matriz de tamaño MAXE x NCAR, por referencia
             listag   vector de NGRUPOS structs (información de grupos generados), por ref.
   Salida:   densidad densidad de los grupos (vector de tamaño NGRUPOS, por referencia)
*******************************************************************************/

void calcular_densidad (float elem[][NCAR], struct lista_grupos *listag, float *densidad)
{
   // PARA COMPLETAR
   // Calcular la densidad de los grupos:
   //        media de las distancia entre todos los elementos del grupo
   //        si el número de elementos del grupo es 0 o 1, densidad = 0
}


/* 4 - Función para relizar el análisis de enfermedades
   Entrada:  listag   vector de NGRUPOS structs (información de grupos generados), por ref.
             enf      enfermedades, una matriz de tamaño MAXE x TENF, por referencia
   Salida:   prob_enf vector de TENF structs (información del análisis realizado), por ref.
*******************************************************************************/

void analizar_enfermedades
(struct lista_grupos *listag, float enf[][TENF], struct analisis *prob_enf)
{
   // PARA COMPLETAR
   // Realizar el análisis de enfermedades en los grupos:
   //        máximo y grupo en el que se da el máximo (para cada enfermedad)
   //        mínimo y grupo en el que se da el mínimo (para cada enfermedad)
}
```

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

#define MAXE     230000   // numero de elementos (muestras)
#define NGRUPOS  100      // numero de clusters
#define NCAR     40       // dimensiones de cada muestra
#define TENF     20       // tipos de enfermedad

#define DELTA    0.01     // convergencia: cambio minimo en un centroide
#define MAXIT    10000    // convergencia: numero de iteraciones maximo


// 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 max, min;    // maximo y minimo de todos los grupos
 int gmax, gmin;    // grupos con los valores maximo y minimo
};
```

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

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

extern void grupo_cercano
       (int nelem, float elem[][NCAR], float cent[][NCAR], int *popul);

extern void calcular_densidad
       (float elem[][NCAR], struct lista_grupos *listag, float *densidad);

extern void analizar_enfermedades
       (struct lista_grupos *listag, float enf[][TENF], struct analisis *prob_enf);
```