1. Paralelismo de bucles

Cuestión 1–1

Según las condiciones de Bernstein, indica el tipo de dependencias de datos existente entre las distintas iteraciones en los casos que se presentan a continuación. Justifica si se puede eliminar o no esa dependencia de datos, eliminándola en caso de que sea posible.

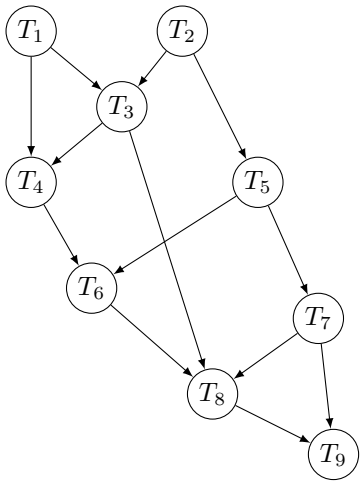
(a)
```
for (i=1;i<N-1;i++) {
  x[i+1] = x[i] + x[i-1];
}
```

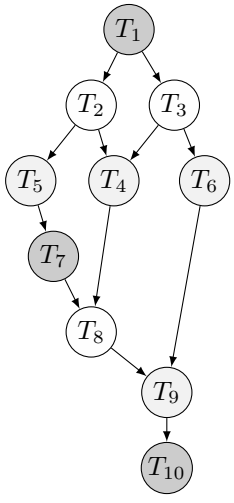
(b)
```
for (i=0;i<N;i++) {
  a[i] = a[i] + y[i];
  x = a[i];
}
```

(c)
```
for (i=N-2;i>=0;i--) {
  x[i] = x[i] + y[i+1];
  y[i] = y[i] + z[i];
}
```

Cuestión 1–2

Dados los siguientes grafos de dependencias de tareas:





T_1, T_7, T_{10}	$\frac{1}{3}n^3$
T_2, T_3, T_8	n^3
T_4, T_5, T_6, T_9	$2n^3$

(a) Para el grafo de la izquierda, indica qué secuencia de nodos del grafo constituye el camino crítico. Calcula la longitud del camino crítico y el grado medio de concurrencia. Nota: no se ofrece información de costes, se puede suponer que todas las tareas tienen el mismo coste.

(b) Repite el apartado anterior para el grafo de la derecha. Nota: en este caso el coste de cada tarea viene dado en flops (para un tamaño de problema n) según la tabla mostrada.

Cuestión 1–3

El siguiente código secuencial implementa el producto de una matriz B de dimensión $N \times N$ por un vector c de dimensión N.

```
void prodmv(double a[N], double c[N], double B[N][N])
{
  int i, j;
  double sum;
  for (i=0; i<N; i++) {
    sum = 0;
    for (j=0; j<N; j++)
      sum += B[i][j] * c[j];
    a[i] = sum;
  }
}
```

(a) Realiza una implementación paralela mediante OpenMP del código dado.

(b) Calcula los costes computacionales en flops de las implementaciones secuencial y paralela, suponiendo que el número de hilos p es un divisor de N.

(c) Calcula el speedup y la eficiencia del código paralelo.

Cuestión 1–4

Dada la siguiente función:

```
double funcion(double A[M][N])
{
  int i,j;
  double suma;
  for (i=0; i<M-1; i++) {
    for (j=0; j<N; j++) {
      A[i][j] = 2.0 * A[i+1][j];
    }
  }
  suma = 0.0;
  for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
      suma = suma + A[i][j];
    }
  }
  return suma;
}
```

(a) Indica su coste teórico (en flops).

(b) Paralelízalo usando OpenMP. ¿Por qué lo haces así? Se valorarán más aquellas soluciones que sean más eficientes.

(c) Indica el speedup que podrá obtenerse con p procesadores suponiendo M y N múltiplos exactos de p.

(d) Indica una cota superior del speedup (cuando p tiende a infinito) si no se paralelizara la parte que calcula la suma (es decir, sólo se paraleliza la primera parte y la segunda se ejecuta secuencialmente).

Cuestión 1–5

Dada la siguiente función:

```
double fun_mat(double a[n][n], double b[n][n])
{
  int i,j,k;
  double aux,s=0.0;
  for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
      aux=0.0;
      s += a[i][j];
      for (k=0; k<n; k++) {
        aux += a[i][k] * a[k][j];
      }
      b[i][j] = aux;
    }
  }
  return s;
}
```

(a) Indica cómo se paralelizaría mediante OpenMP cada uno de los tres bucles. ¿Cuál de las tres formas de paralelizar será la más eficiente y por qué?

(b) Suponiendo que se paraleliza el bucle más externo, indica los costes a priori secuencial y paralelo, en flops, y el speedup suponiendo que el número de hilos (y procesadores) coincide con n.

(c) Añade las líneas de código necesarias para que se muestre en pantalla el número de iteraciones que ha realizado el hilo 0, suponiendo que se paraleliza el bucle más externo.

Cuestión 1–6

Implementa un programa paralelo utilizando OpenMP que cumpla los siguientes requisitos:

- Pida por teclado un número entero positivo n.
- Calcule en paralelo la suma de los primeros n números naturales, utilizando para ello una distribución dinámica que reparta los números a sumar de 2 en 2, siendo 6 el número de hilos usado.
- Al final del programa deberá imprimir en pantalla el identificador del hilo que ha sumado el último número (n) y la suma total calculada.

Cuestión 1–7

Se quiere paralelizar de forma eficiente la siguiente función mediante OpenMP.

```
#define EPS 1e-9
#define N 128
int fun(double a[N][N], double b[], double x[], int n, int nMax)
{
  int i, j, k;
  double err=100, aux[N];

  for (i=0;i<n;i++)
     aux[i]=0.0;

  for (k=0;k<nMax && err>EPS;k++) {
     err=0.0;
     for (i=0;i<n;i++) {
        x[i]=b[i];
        for (j=0;j<i;j++)
           x[i]-=a[i][j]*aux[j];
```

```
        for (j=i+1;j<n;j++)
           x[i]-=a[i][j]*aux[j];
        x[i]/=a[i][i];
        err+=fabs(x[i]-aux[i]);
      }
      for (i=0;i<n;i++)
        aux[i]=x[i];
  }
  return k<nMax;
}
```

(a) Paralelízala de forma eficiente.

(b) Calcula el coste computacional de una iteración del bucle k. Calcula el coste computacional de la versión paralela (asumiendo que se divide el número de iteraciones de forma exacta entre el número de hilos) y el speed-up.

Cuestión 1–8

Dada la siguiente función:

```
#define N 6000
#define PASOS 6

double funcion1(double A[N][N], double b[N], double x[N])
{
  int i, j, k, n=N, pasos=PASOS;
  double max=-1.0e308, q, s, x2[N];
  for (k=0;k<pasos;k++) {
    q=1;
    for (i=0;i<n;i++) {
      s = b[i];
      for (j=0;j<n;j++)
        s -= A[i][j]*x[j];
      x2[i] = s;
      q *= s;
    }
    for (i=0;i<n;i++)
      x[i] = x2[i];
    if (max<q)
      max = q;
  }
  return max;
}
```

(a) Paraleliza el código usando OpenMP. ¿Por qué lo haces así? Se valorarán más aquellas soluciones que sean más eficientes.

(b) Indica el coste teórico (en flops) que tendría una iteración del bucle `k` del código secuencial.

(c) Considerando una única iteración del bucle `k` (`PASOS`=1), indica el speedup y la eficiencia que podrá obtenerse con p hilos, suponiendo que hay tantos núcleos/procesadores como hilos y que `N` es un múltiplo exacto de p.

Cuestión 1–9

Dada la siguiente función:

```
void func(double A[M][P], double B[P][N], double C[M][N], double v[M]) {
   int i, j, k;
   double mf, val;
   for (i=0; i<M; i++) {
      mf = 0;
      for (j=0; j<N; j++) {
         val = 2.0*C[i][j];
         for (k=0; k<i; k++) {
            val += A[i][k]*B[k][j];
         }
         C[i][j] = val;
         if (val<mf) mf = val;
      }
      v[i] += mf;
   }
}
```

(a) Haz una versión paralela basada en la paralelización del bucle `i`.

(b) Haz una versión paralela basada en la paralelización del bucle `j`.

(c) Calcula el tiempo de ejecución secuencial a priori de una sola iteración del bucle `i`, así como el tiempo de ejecución secuencial de la función completa. Supón que el coste de una comparación de números en coma flotante es 1 flop.

(d) Indica si habría un buen equilibrio de carga si se usa la cláusula `schedule(static)` en la paralelización del primer apartado. Razona la respuesta.

Cuestión 1–10

Dada la siguiente función:

```
double cuad_mat(double a[N][N], double b[N][N])
{
   int i,j,k;
   double aux, s=0.0;
   for (i=0; i<N; i++) {
      for (j=0; j<N; j++) {
         aux = 0.0;
         for (k=i; k<N; k++)
            aux += a[i][k] * a[k][j];
         b[i][j] = aux;
         s += aux*aux;
      }
   }
   return s;
}
```

(a) Paraleliza el código anterior de forma eficiente mediante OpenMP. De las posibles planificaciones, ¿cuáles de ellas podrían ser las más eficientes? Justifica la respuesta.

(b) Calcula el coste del algoritmo secuencial en flops.

Cuestión 1–11

Dada la siguiente función:

```
double f(double A[N][N], double B[N][N], double vs[N], double bmin) {
  int i, j;
```

```
  double x, y, aux, stot=0;
  for (i=0; i<N; i++) {
    aux = 0;
    for (j=0; j<N; j++) {
      x = A[i][j]*A[i][j]/2.0;
      A[i][j] = x;
      aux += x;
    }
    for (j=i; j<N; j++) {
      if (B[i][j]<bmin) y = bmin;
      else              y = B[i][j];
      B[i][j] = 1.0/y;
    }
    vs[i] = aux;
    stot += vs[i];
  }
  return stot;
}
```

(a) Paraleliza (eficientemente) el bucle `i` mediante OpenMP.

(b) Paraleliza (eficientemente) los dos bucles `j` mediante OpenMP.

(c) Calcula el coste secuencial del código original.

(d) Suponiendo que paralelizamos solo el primer bucle `j`, calcula el coste paralelo de dicha versión . Obtén el speedup y la eficiencia en el caso de que se disponga de `N` procesadores.

2. Regiones paralelas

Cuestión 2–1

Dada la siguiente función, que busca un valor en un vector, paralelízala usando OpenMP. Al igual que la función de partida, la función paralela deberá terminar la búsqueda tan pronto como se encuentre el elemento buscado.

```
int busqueda(int x[], int n, int valor)
{
   int encontrado=0, i=0;
   while (!encontrado && i<n) {
      if (x[i]==valor) encontrado=1;
      i++;
   }
   return encontrado;
}
```

Cuestión 2–2

Dado un vector v de n elementos, la siguiente función calcula su 2-norma $\|v\|$, definida como:

$$\|v\| = \sqrt{\sum_{i=1}^{n} v_i^2}$$

```
double norma(double v[], int n)
{
```

```
   int i;
   double r=0;
   for (i=0; i<n; i++)
      r += v[i]*v[i];
   return sqrt(r);
}
```

(a) Paralelizar la función anterior mediante OpenMP, siguiendo el siguiente esquema:

- En una primera fase, se quiere que cada hilo calcule la suma de cuadrados de un bloque de n/p elementos del vector v (siendo p el número de hilos). Cada hilo dejará el resultado en la posición correspondiente de un vector `sumas` de p elementos. Se puede asumir que el vector `sumas` ya ha sido creado (aunque no inicializado).
- En una segunda fase, uno de los hilos calculará la norma del vector, a partir de las sumas parciales almacenadas en el vector `sumas`.

(b) Paralelizar la función de partida mediante OpenMP, usando otra aproximación distinta de la del apartado anterior.

(c) Calcular el coste a priori del algoritmo secuencial de partida. Razonar cuál sería el coste del algoritmo paralelo del apartado a, y el speedup obtenido.

Cuestión 2–3

Dada la siguiente función:

```
void f(int n, double a[], double b[])
{
   int i;
   for (i=0; i<n; i++) {
      b[i]=cos(a[i]);
   }
}
```

Paralelízala, haciendo además que cada hilo escriba un mensaje indicando su número de hilo y cuántas iteraciones ha procesado. Se quiere mostrar un solo mensaje por cada hilo.

Cuestión 2–4

Dada la siguiente función:

```
void normaliza(double A[N][N])
{
  int i,j;
  double suma=0.0,factor;
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      suma = suma + A[i][j]*A[i][j];
    }
  }
  factor = 1.0/sqrt(suma);
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      A[i][j] = factor*A[i][j];
    }
  }
}
```

(a) Paralelízala con OpenMP usando dos regiones paralelas.

(b) Paralelízala con OpenMP usando una única región paralela que englobe a todos los bucles. En este caso, ¿tendría sentido utilizar la cláusula `nowait`?

Cuestión 2–5

Dada la siguiente función:

```
double ej(double x[M], double y[N], double A[M][N])
{
   int i,j;
   double aux,s=0.0;
   for (i=0; i<M; i++)
      x[i] = x[i]*x[i];
   for (i=0; i<N; i++)
      y[i] = 1.0+y[i];
   for (i=0; i<M; i++)
      for (j=0; j<N; j++) {
         aux = x[i]-y[j];
         A[i][j] = aux;
         s += aux;
      }
   return s;
}
```

(a) Paralelízala eficientemente mediante OpenMP, usando para ello una sola región paralela.

(b) Calcula el número de flops de la función inicial y de la función paralelizada.

(c) Determina el speedup y la eficiencia.

Cuestión 2–6

Paraleliza el siguiente fragmento de código mediante secciones de OpenMP. El segundo argumento de las funciones `fun1`, `fun2` y `fun3` es de entrada-salida, es decir, estas funciones utilizan y modifican el valor de `a`.

```
int n=...;
double a,b[3];

a = -1.8;
fun1(n,&a);
b[0] = a;
a = 3.2;
fun2(n,&a);
b[1] = a;
a = 0.25;
fun3(n,&a);
b[2] = a;
```

Cuestión 2–7

Dada la siguiente función:

```
void func(double a[],double b[],double c[],double d[])
{
  f1(a,b);
  f2(b,b);
```

```
  f3(c,d);
  f4(d,d);
  f5(a,a,b,c,d);
}
```

El primer argumento de todas las funciones usadas es de salida y el resto de argumentos son argumentos de entrada. Por ejemplo, `f1(a,b)` es una función que a partir del vector `b` modifica el vector `a`.

(a) Dibuja el grafo de dependencias de tareas e indica al menos 2 tipos diferentes de dependencias que aparezcan en este problema.

(b) Paraleliza la función mediante directivas OpenMP.

(c) Suponiendo que todas las funciones tienen el mismo coste y que se dispone de un número de procesadores arbitrario, ¿cuál será el speedup máximo posible? ¿Se podría mejorar este speedup utilizando replicación de datos?

Cuestión 2–8

En la siguiente función, `T1`, `T2`, `T3` modifican `x`, `y`, `z`, respectivamente.

```
double f(double x[], double y[], double z[], int n)
{
   int i, j;
   double s1, s2, a, res;

   T1(x,n);    /* Tarea T1 */
   T2(y,n);    /* Tarea T2 */
   T3(z,n);    /* Tarea T3 */
   /* Tarea T4 */
   for (i=0; i<n; i++) {
      s1=0;
      for (j=0; j<n; j++) s1+=x[i]*y[i];
      for (j=0; j<n; j++) x[i]*=s1;
   }
   /* Tarea T5 */
   for (i=0; i<n; i++) {
      s2=0;
      for (j=0; j<n; j++) s2+=y[i]*z[i];
      for (j=0; j<n; j++) z[i]*=s2;
   }
   /* Tarea T6 */
   a=s1/s2;
   res=0;
   for (i=0; i<n; i++) res+=a*z[i];
   return res;
}
```

(a) Dibuja el grafo de dependencia de las tareas.

(b) Realiza una paralelización mediante OpenMP a nivel de tareas (no de bucles), basándote en el grafo de dependencias.

(c) Indica el coste a priori del algoritmo secuencial, el del algoritmo paralelo y el speedup resultante. Supón que el coste de las tareas 1, 2 y 3 es de $2n^2$ flops cada una.

Cuestión 2–9

Dado el siguiente fragmento de código:

```
minx = minimo(x,n);       /* T1 */
maxx = maximo(x,n);       /* T2 */
calcula_z(z,minx,maxx,n); /* T3 */
calcula_y(y,x,n);         /* T4 */
calcula_x(x,y,n);         /* T5 */
calcula_v(v,z,x);         /* T6 */
```

(a) Dibuja el grafo de dependencias de las tareas, teniendo en cuenta que las funciones `minimo` y `maximo` no modifican sus argumentos, mientras que las demás funciones modifican sólo su primer argumento.

(b) Paraleliza el código mediante OpenMP.

(c) Si el coste de las tareas es de n flops, excepto el de la tarea 4 que es de $2n$ flops, indica la longitud del camino crítico y el grado medio de concurrencia. Obtén el speedup y la eficiencia de la implementación del apartado anterior, si se ejecutara con 5 procesadores.

Cuestión 2–10

Se quiere paralelizar el siguiente programa mediante OpenMP, donde `genera` es una función previamente definida en otro lugar.

```
double fun1(double a[],int n,
            int v0)
{
  int i;
  a[0] = v0;
  for (i=1;i<n;i++)
    a[i] = genera(a[i-1],i);
}
```

```
double compara(double x[],double y[],int n)
{
  int i;
  double s=0;
  for (i=0;i<n;i++)
    s += fabs(x[i]-y[i]);
  return s;
}
```

```
/* fragmento del programa principal (main) */
int i, n=10;
double a[10], b[10], c[10], x=5, y=7, z=11, w;
fun1(a,n,x);           /* T1 */
fun1(b,n,y);           /* T2 */
fun1(c,n,z);           /* T3 */
x = compara(a,b,n);    /* T4 */
y = compara(a,c,n);    /* T5 */
z = compara(c,b,n);    /* T6 */
w = x+y+z;             /* T7 */
printf("w:%f\n", w);
```

(a) Paraleliza el código de forma eficiente a nivel de bucles.

(b) Dibuja el grafo de dependencias de tareas, según la numeración de tareas indicada en el código.

(c) Paraleliza el código de forma eficiente a nivel de tareas, a partir del grafo de dependencias anterior.

(d) Obtén el tiempo secuencial (asume que una llamada a las funciones `genera` y `fabs` cuesta 1 flop) y el tiempo paralelo para cada una de las dos versiones asumiendo que hay 3 procesadores. Calcular el speed-up en cada caso.

Cuestión 2–11

Paraleliza mediante OpenMP el siguiente fragmento de código, donde `f` y `g` son dos funciones que toman 3 argumentos de tipo `double` y devuelven un `double`, y `fabs` es la función estándar que devuelve el valor absoluto de un `double`.

```
double x,y,z,w=0.0;
double x0=1.0,y0=3.0,z0=2.0;     /* punto inicial */
double dx=0.01,dy=0.01,dz=0.01;  /* incrementos */

x=x0;y=y0;z=z0;    /* busca en x */
while (fabs(f(x,y,z))<fabs(g(x0,y0,z0))) x += dx;
w += (x-x0);

x=x0;y=y0;z=z0;    /* busca en y */
while (fabs(f(x,y,z))<fabs(g(x0,y0,z0))) y += dy;
w += (y-y0);

x=x0;y=y0;z=z0;    /* busca en z */
while (fabs(f(x,y,z))<fabs(g(x0,y0,z0))) z += dz;
w += (z-z0);

printf("w = %g\n",w);
```

Cuestión 2–12

Teniendo en cuenta la definición de las siguientes funciones:

```
/* producto matricial C = A*B */
void matmult(double A[N][N],
       double B[N][N],double C[N][N])
{
  int i,j,k;
  double suma;
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      suma = 0.0;
      for (k=0; k<N; k++) {
        suma = suma + A[i][k]*B[k][j];
      }
      C[i][j] = suma;
    }
  }
}
```

```
/* simetriza una matriz como A+A' */
void simetriza(double A[N][N])
{
  int i,j;
  double suma;
  for (i=0; i<N; i++) {
    for (j=0; j<=i; j++) {
      suma = A[i][j]+A[j][i];
      A[i][j] = suma;
      A[j][i] = suma;
    }
  }
}
```

se pretende paralelizar el siguiente código:

```
matmult(X,Y,C1);    /* T1 */
matmult(Y,Z,C2);    /* T2 */
matmult(Z,X,C3);    /* T3 */
simetriza(C1);      /* T4 */
simetriza(C2);      /* T5 */
matmult(C1,C2,D1);  /* T6 */
matmult(D1,C3,D);   /* T7 */
```

(a) Realiza una paralelización basada en los bucles.

(b) Dibuja el grafo de dependencias de tareas, considerando en este caso que las tareas son cada una de las llamadas a `matmult` y `simetriza`. Indica cuál es el grado máximo de concurrencia, la longitud del camino crítico y el grado medio de concurrencia. Nota: para determinar estos últimos valores, es necesario obtener el coste en flops de ambas funciones.

(c) Realiza la paralelización basada en secciones, a partir del grafo de dependencias anterior.

Cuestión 2–13

Dada la siguiente función:

```
void updatemat(double A[N][N])
{
  int i,j;
  double s[N];
  for (i=0; i<N; i++) {     /* suma de filas */
    s[i] = 0.0;
    for (j=0; j<N; j++)
      s[i] += A[i][j];
  }
  for (i=1; i<N; i++)       /* suma prefija */
    s[i] += s[i-1];
  for (j=0; j<N; j++) {     /* escalado de columnas */
    for (i=0; i<N; i++)
      A[i][j] *= s[j];
  }
}
```

(a) Indica el coste teórico (en flops) de la función proporcionada.

(b) Paralelízala con OpenMP con una única región paralela.

(c) Indica el speedup que podrá obtenerse con p procesadores suponiendo que N es múltiplo exacto de p.

Cuestión 2–14

Dada la siguiente función:

```
double calcula()
{
  double A[N][N],B[N][N],a,b,x,y,z;

  rellena(A,B);             /* T1 */
  a = calculos(A);          /* T2 */
  b = calculos(B);          /* T3 */
  x = suma_menores(B,a);    /* T4 */
  y = suma_en_rango(B,a,b); /* T5 */
  z = x + y;                /* T6 */
  return z;
}
```

La función `rellena` recibe dos matrices y las rellena con valores generados internamente. Los parámetros del resto de funciones son sólo de entrada (no se modifican). Las funciones `rellena` y `suma_en_rango` tienen un coste de $2n^2$ flops cada una ($n =$ `N`), mientras que el coste de cada una de las otras funciones es n^2 flops.

(a) Dibuja el grafo de dependencias e indica su grado máximo de concurrencia, un camino crítico y su longitud y el grado medio de concurrencia.

(b) Paraleliza la función con OpenMP.

(c) Calcula el tiempo de ejecución secuencial, el tiempo de ejecución paralelo, el speed-up y la eficiencia del código del apartado anterior, suponiendo que se trabaja con 3 hilos.

Cuestión 2–15

Se quiere paralelizar el siguiente código de procesado de imágenes, que recibe como entrada 4 imágenes similares (por ejemplo, fotogramas de un vídeo `f1`, `f2`, `f3`, `f4`) y devuelve dos imágenes resultado (`r1`, `r2`). Los píxeles de la imagen se representan como números en coma flotante (`image` es un nuevo tipo de datos consistente en una matriz de `N`×`M` `double`s).

```
typedef double image[N][M];

void procesa(image f1,image f2,image f3,image f4,image r1,image r2)
{
  image d1,d2,d3;
  difer(f2,f1,d1);          /* Tarea 1 */
  difer(f3,f2,d2);          /* Tarea 2 */
  difer(f4,f3,d3);          /* Tarea 3 */
  suma(d1,d2,d3,r1);        /* Tarea 4 */
  difer(f4,f1,r2);          /* Tarea 5 */
}
```

```
void difer(image a,image b,image d)
{
  int i,j;
  for (i=0;i<N;i++)
    for (j=0;j<M;j++)
      d[i][j] = fabs(a[i][j]-b[i][j]);
}
```

```
void suma(image a,image b,image c,image s)
{
  int i,j;
  for (i=0;i<N;i++)
    for (j=0;j<M;j++)
      s[i][j] = a[i][j]+b[i][j]+c[i][j];
}
```

(a) Dibuja el grafo de dependencias de tareas, e indica cuál sería el grado máximo y medio de concurrencia, teniendo en cuenta el coste en flops (supón que `fabs` no realiza ningún flop).

(b) Paraleliza la función `procesa` mediante OpenMP, sin modificar `difer` y `suma`.

Cuestión 2–16

En la siguiente función, ninguna de las funciones llamadas (`A`,`B`,`C`,`D`) modifica sus parámetros:

```
double calculos_matriciales(double mat[n][n])
{
  double x,y,z,aux,total;
  x = A(mat);          /* tarea A, coste: 3 n^2         */
  aux = B(mat);        /* tarea B, coste: n^2           */
  y = C(mat,aux);      /* tarea C, coste: n^2           */
  z = D(mat);          /* tarea D, coste: 2 n^2         */
  total = x + y + z;   /* tarea E (calcula tú su coste) */
  return total;
}
```

(a) Dibuja su grafo de dependencias e indica el grado máximo de concurrencia, la longitud del camino crítico indicando un camino crítico y el grado medio de concurrencia.

(b) Paralelízala con OpenMP.

(c) Calcula el tiempo secuencial en flops. Asumiendo que se va a ejecutar con 2 hilos, calcula el tiempo paralelo, el speedup y la eficiencia, en el mejor de los casos.

(d) Modifica el código paralelo para que se muestre por pantalla (una sola vez) el número de hilos con que se ejecute cada vez y el tiempo de ejecución utilizado en segundos.

Cuestión 2–17

Dada la siguiente función:

```
double funcion(double A[M][N], double maximo, double pf[])
{
   int i,j,j2;
   double a,x,y;
   x = 0;
   for (i=0; i<M; i++) {
      y = 1;
      for (j=0; j<N; j++) {
         a = A[i][j];
         if (a>maximo) a = 0;
         x += a;
      }
      for (j2=1; j2<i; j2++) {
         y *= A[i][j2-1]-A[i][j2];
      }
      pf[i] = y;
   }
   return x;
}
```

(a) Haz una versión paralela basada en la paralelización del bucle `i` con OpenMP.

(b) Haz otra versión paralela basada en la paralelización de los bucles `j` y `j2` (de forma eficiente para cualquier número de hilos).

(c) Calcula el coste (tiempo de ejecución) del código secuencial.

(d) Para cada uno de los tres bucles, justifica si cabe esperar diferencias de prestaciones dependiendo de la planificación empleada al paralelizar el bucle. Si es así, indica qué planificaciones serían mejor para el bucle correspondiente.

Cuestión 2–18

Se desea paralelizar la siguiente función, donde `lee_datos` modifica sus tres argumentos y `f5` lee y escribe sus dos primeros argumentos. El resto de funciones no modifican sus argumentos.

```
void funcion() {
  double x,y,z,a,b,c,d,e;
  int n;
  n = lee_datos(&x,&y,&z);     /* Tarea 1 (n flops)    */
  a = f2(x,n);                 /* Tarea 2 (2n flops)   */
  b = f3(y,n);                 /* Tarea 3 (2n flops)   */
  c = f4(z,a,n);               /* Tarea 4 (n^2 flops)  */
  d = f5(&x,&y,n);             /* Tarea 5 (3n^2 flops) */
  e = f6(z,b,n);               /* Tarea 6 (n^2 flops)  */
  escribe_resultados(c,d,e);   /* Tarea 7 (n flops)    */
}
```

(a) Dibuja el grafo de dependencias de las diferentes tareas que componen la función.

(b) Paraleliza la función eficientemente con OpenMP.

(c) Obtén el speedup y la eficiencia si empleamos 3 procesadores.

(d) A partir de los costes de cada tarea reflejados en el código de la función, obtén la longitud del camino crítico y el grado medio de concurrencia.

Cuestión 2–19

Dada la siguiente función, donde sabemos que todas las funciones a las que se llama modifican solo el vector que reciben como primer argumento:

```
double f(double x[], double y[], double z[], double v[], double w[]) {
   double r1, res;
   A(x,v);              /* Tarea A. Coste de 2*n^2 flops */
   B(y,v,w);            /* Tarea B. Coste de   n   flops */
   C(w,v);              /* Tarea C. Coste de   n^2 flops */
   r1=D(z,v);           /* Tarea D. Coste de 2*n^2 flops */
   E(x,v,w);            /* Tarea E. Coste de   n^2 flops */
   res=F(z,r1);         /* Tarea F. Coste de 3*n   flops */
   return res;
}
```

(a) Dibuja el grafo de dependencias. Identifica un camino crítico e indica su longitud. Calcula el grado medio de concurrencia.

(b) Implementa una versión paralela eficiente de la función.

(c) Suponiendo que el código del apartado anterior se ejecuta con 2 hilos, calcula el tiempo de ejecución paralelo, el speed-up y la eficiencia, en el mejor de los casos. Razona la respuesta.

Cuestión 2–20

En la siguiente función ninguna de las funciones a las que llama modifican sus parámetros.

```
int ejercicio(double v[n],double x)
{
  int i,j,k=0;
  double a,b,c;
  a = tarea1(v,x);   /* tarea 1, coste n flops */
  b = tarea2(v,a);   /* tarea 2, coste n flops */
  c = tarea3(v,x);   /* tarea 3, coste 4n flops */
  x = x + a + b + c;   /* tarea 4 */
  for (i=0; i<n; i++) {   /* tarea 5 */
    j = f(v[i],x);   /* cada llamada a esta función cuesta 6 flops */
    if (j>0 && j<4) k++;
  }
  return k;
}
```

(a) Calcula el tiempo de ejecución secuencial.

(b) Dibuja el grafo de dependencias a nivel de tareas (considerando la tarea 5 como indivisible) e indica el grado máximo de concurrencia, la longitud del camino crítico y el grado medio de concurrencia.

(c) Paralelízala de forma eficiente usando una sola región paralela. Aparte de realizar en paralelo aquellas tareas que se puedan, paraleliza también el bucle de la tarea 5.

(d) Suponiendo que se ejecuta con 6 hilos (y que n es un múltiplo exacto de 6), calcula el tiempo de ejecución paralelo, el speed-up y la eficiencia.

Cuestión 2–21

```
void matmult(double A[N][N],
       double B[N][N],double C[N][N]) {
  int i,j,k;
  double sum;
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      sum = 0.0;
      for (k=0; k<N; k++) {
        sum += A[i][k]*B[k][j];
      }
      C[i][j] = sum;
    }
  }
}
```

```
void normalize(double A[N][N]) {
  int i,j;
  double sum=0.0,factor;
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      sum += A[i][j]*A[i][j];
    }
  }
  factor = 1.0/sqrt(sum);
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      A[i][j] *= factor;
    }
  }
}
```

Dada la definición de las funciones anteriores, se pretende paralelizar el siguiente código:

```
matmult(A,B,R1);    /* T1 */
matmult(C,D,R2);    /* T2 */
normalize(R1);      /* T3 */
normalize(R2);      /* T4 */
matmult(A,R2,M1);   /* T5 */
matmult(B,R2,M2);   /* T6 */
matmult(C,R1,M3);   /* T7 */
matmult(D,R1,M4);   /* T8 */
```

(a) Dibuja el grafo de dependencias de tareas. Indica cuál es la longitud del camino crítico y el grado medio de concurrencia. Nota: para determinar estos últimos valores, es necesario obtener el coste en flops de ambas funciones. Asumir que `sqrt` cuesta 5 flops.

(b) Realiza la paralelización basada en secciones, a partir del grafo de dependencias anterior.

Cuestión 2–22

Dada la siguiente función:

```
double sumar(double A[N][M])
{
   double suma=0, maximo;
   int i,j;

   for (i=0; i<N; i++) {
      maximo=0;
      for (j=0; j<M; j++) {
         if (A[i][j]>maximo) maximo = A[i][j];
      }
      for (j=0; j<M; j++) {
         if (A[i][j]>0.0) {
            A[i][j] = A[i][j]/maximo;
            suma = suma + A[i][j];
         }
      }
   }
   return suma;
}
```

(a) Paraleliza la función de forma eficiente mediante OpenMP.

(b) Indica su coste paralelo teórico (en flops), asumiendo que `N` es múltiplo del número de hilos. Para evaluar el coste considera el caso peor, es decir, que todas las comparaciones son ciertas. Además, supon que el coste de comparar dos números reales es 1 *flop*.

(c) Modifica el código para que cada hilo muestre un único mensaje con su número de hilo y el número de elementos que ha sumado.

Cuestión 2–23

Sea el siguiente código:

```
double a,b,c,e,d,f;
T1(&a,&b); // Coste: 10 flops
c=T2(a);   // Coste: 15 flops
c=T3(c);   // Coste:  8 flops
d=T4(b);   // Coste: 20 flops
e=T5(c);   // Coste: 30 flops
f=T6(c);   // Coste: 35 flops
b=T7(c);   // Coste: 30 flops
```

(a) Obtén el grafo de dependencias y explica qué tipo de dependencias ocurren entre T_2 y T_3 y entre T_4 y T_7, en caso de que las haya.

(b) Calcula la longitud del camino crítico, e indica las tareas que lo forman.

(c) Implementa una versión paralela lo más eficiente posible del código anterior mediante secciones, empleando una única región paralela.

(d) Calcula el speedup y la eficiencia si empleáramos 4 hilos para ejecutar el código paralelizado en el apartado anterior.

3. Sincronización

Cuestión 3–1

Sea el siguiente código que permite ordenar un vector `v` de `n` números reales ascendentemente:

```
int ordenado = 0;
double a;
while( !ordenado ) {
  ordenado = 1;
  for( i=0; i<n-1; i+=2 ) {
    if( v[i]>v[i+1] ) {
      a = v[i];
      v[i] = v[i+1];
      v[i+1] = a;
      ordenado = 0;
    }
  }
  for( i=1; i<n-1; i+=2 ) {
    if( v[i]>v[i+1] ) {
      a = v[i];
      v[i] = v[i+1];
      v[i+1] = a;
      ordenado = 0;
    }
```

```
    }
  }
```

(a) Introducir las directivas OpenMP que permitan ejecutar este código en paralelo.

(b) Modificar el código para contabilizar el número de intercambios que se producen, es decir, el número de veces que se entra en cualquiera de las dos estructuras `if`.

Cuestión 3–2

Dada la función:

```
void f(int n, double v[], double x[], int ind[])
{
   int i;
   for (i=0; i<n; i++) {
      x[ind[i]] = max(x[ind[i]],f2(v[i]));
   }
}
```

Paralelizar la función, teniendo en cuenta que `f2` es una función muy costosa. Se valorará que la solución aportada sea eficiente.

Nota. Se asume que `f2` no tiene efectos laterales y su resultado sólo depende de su argumento de entrada. El tipo de retorno de la función `f2` es `double`. La función `max` devuelve el máximo de dos números.

Cuestión 3–3

Dada la siguiente función, la cual busca un valor en un vector

```
int buscar(int x[], int n, int valor)
{
   int i, pos=-1;

   for (i=0; i<n; i++)
      if (x[i]==valor)
         pos=i;

   return pos;
}
```

Se pide paralelizarla mediante OpenMP. En caso de varias ocurrencias del valor en el vector, el algoritmo paralelo debe devolver lo mismo que el secuencial.

Cuestión 3–4

La infinito-norma de una matriz $A \in \mathbb{R}^{n \times n}$ se define como el máximo de las sumas de los valores absolutos de los elementos de cada fila:

$$\|A\|_\infty = \max_{i=0,\ldots,n-1} \left\{ \sum_{j=0}^{n-1} |a_{i,j}| \right\}$$

El siguiente código secuencial implementa dicha operación para el caso de una matriz cuadrada.

```
#include <math.h>
#define DIMN 100

double infNorm(double A[DIMN][DIMN], int n)
{
  int i,j;
```

```
  double s,norm=0;

  for (i=0; i<n; i++) {
    s = 0;
    for (j=0; j<n; j++)
      s += fabs(A[i][j]);
    if (s>norm)
      norm = s;
  }
  return norm;
}
```

(a) Realiza una implementación paralela mediante OpenMP de dicho algoritmo. Justifica la razón por la que introduces cada cambio.

(b) Calcula el coste computacional (en flops) de la versión original secuencial y de la versión paralela desarrollada.
Nota: Se puede asumir que la dimensión de la matriz n es un múltiplo exacto del número de hilos p. Se puede asumir que el coste de la función `fabs` es de 1 flop.

(c) Calcula el speedup y la eficiencia del código paralelo ejecutado en p procesadores.

Cuestión 3–5

Dada la siguiente función, que calcula el producto de los elementos del vector `v`:

```
double prod(double v[], int n)
{
  double p=1;
  int i;
  for (i=0;i<n;i++)
    p *= v[i];
  return p;
}
```

Implementa dos funciones paralelas:

(a) Utilizando reducción.

(b) Sin utilizar reducción.

Cuestión 3–6

Se quiere paralelizar de forma eficiente la siguiente función mediante OpenMP.

```
int cmp(int n, double x[], double y[], int z[])
{
  int i, v, equal=0;
  double aux;
  for (i=0; i<n; i++) {
    aux = x[i] - y[i];
    if (aux > 0) v = 1;
    else if (aux < 0) v = -1;
    else v = 0;
    z[i] = v;
    if (v == 0) equal++;
  }
  return equal;
}
```

(a) Paralelízala utilizando construcciones de tipo `parallel for`.

(b) Paralelízala sin usar ninguna de las siguientes primitivas: `for`, `section`, `reduction`.

Cuestión 3–7

Dado el siguiente fragmento de código, donde el vector de índices `ind` contiene valores enteros entre 0 y $m-1$ (siendo m la dimensión de `x`), posiblemente con repeticiones:

```
for (i=0; i<n; i++) {
   s = 0;
   for (j=0; j<i; j++) {
      s += A[i][j]*b[j];
   }
   c[i] = s;
   x[ind[i]] += s;
}
```

(a) Realiza una implementación paralela mediante OpenMP, en la que se reparten las iteraciones del bucle externo.

(b) Realiza una implementación paralela mediante OpenMP, en la que se reparten las iteraciones del bucle interno.

(c) Para la implementación del apartado (a), indica si cabe esperar que haya diferencias de prestaciones dependiendo de la planificación empleada. Si es así, ¿qué planificaciones serían mejores y por qué?

Cuestión 3–8

La siguiente función normaliza los valores de un vector de números reales positivos de forma que los valores finales queden entre 0 y 1, utilizando el máximo y el mínimo.

```
void normalize(double *a, int n)
{
  double mx, mn, factor;
  int i;

  mx = a[0];
  for (i=1;i<n;i++) {
    if (mx<a[i]) mx=a[i];
  }
  mn = a[0];
  for (i=1;i<n;i++) {
    if (mn>a[i]) mn=a[i];
  }
  factor = mx-mn;
  for (i=0;i<n;i++) {
    a[i]=(a[i]-mn)/factor;
  }
}
```

(a) Paraleliza el programa con OpenMP de la manera más eficiente posible, mediante una única región paralela. Supóngase que el valor de `n` es muy grande y que se quiere que la paralelización funcione eficientemente para un número arbitrario de hilos.

(b) Incluye el código necesario para que se imprima una sola vez el número de hilos utilizados.

Cuestión 3–9

Dada la siguiente función:

```
int funcion(int n, double v[])
{
  int i,pos_max=-1;
  double suma,norma,aux,max=-1;

  suma = 0;
  for (i=0;i<n;i++)
    suma = suma + v[i]*v[i];
  norma = sqrt(suma);

  for (i=0;i<n;i++)
    v[i] = v[i] / norma;

  for (i=0;i<n;i++) {
    aux = v[i];
    if (aux < 0)  aux = -aux;
    if (aux > max) {
      pos_max = i;  max = aux;
    }
  }
  return pos_max;
}
```

(a) Paralelízala con OpenMP, usando una única región paralela.

(b) ¿Tendría sentido poner una cláusula `nowait` a alguno de los bucles? ¿Por qué? Justifica cada bucle separadamente.

(c) ¿Qué añadirías para garantizar que en todos los bucles las iteraciones se reparten de 2 en 2 entre los hilos?

Cuestión 3–10

La siguiente función procesa una serie de transferencias bancarias. Cada transferencia tiene una cuenta origen, una cuenta destino y una cantidad de dinero que se mueve de la cuenta origen a la cuenta destino. La función actualiza la cantidad de dinero de cada cuenta (array `saldos`) y además devuelve la cantidad máxima que se transfiere en una sola operación.

```
double transferencias(double saldos[], int origenes[],
       int destinos[], double cantidades[], int n)
{
   int i, i1, i2;
   double dinero, maxtransf=0;

   for (i=0; i<n; i++) {
      /* Procesar transferencia i: La cantidad transferida es
       * cantidades[i], que se mueve de la cuenta origenes[i]
       * a la cuenta destinos[i]. Se actualizan los saldos de
       * ambas cuentas y la cantidad maxima */
      i1 = origenes[i];
      i2 = destinos[i];
      dinero = cantidades[i];
      saldos[i1] -= dinero;
      saldos[i2] += dinero;
      if (dinero>maxtransf) maxtransf = dinero;
   }
```

```
   return maxtransf;
}
```

(a) Paraleliza la función de forma eficiente mediante OpenMP.

(b) Modifica la solución del apartado anterior para que se imprima el índice de la transferencia con más dinero.

Cuestión 3–11

Sea la siguiente función:

```
double funcion(double A[N][N],double B[N][N])
{
  int i,j;
  double aux, maxi;
  for (i=1; i<N; i++) {
    for (j=0; j<N; j++) {
      A[i][j] = 2.0+A[i-1][j];
    }
  }
  for (i=0; i<N-1; i++) {
    for (j=0; j<N-1; j++) {
      B[i][j] = A[i+1][j]*A[i][j+1];
    }
  }
  maxi = 0.0;
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      aux = B[i][j]*B[i][j];
      if (aux>maxi) maxi = aux;
    }
  }
  return maxi;
}
```

(a) Paraleliza el código anterior mediante OpenMP. Explica las decisiones que tomes. Se valorarán más aquellas soluciones que sean más eficientes.

(b) Calcula el coste secuencial, el coste paralelo, el speedup y la eficiencia que podrán obtenerse con p procesadores suponiendo que N es múltiplo de p.

Cuestión 3–12

La siguiente función proporciona todas las posiciones de fila y columna en las que se encuentra repetido el valor máximo de una matriz:

```
int funcion(double A[N][N],double posiciones[][2])
{
  int i,j,k=0;
  double maximo;
  /* Calculamos el máximo */
  maximo = A[0][0];
  for (i=0;i<N;i++) {
    for (j=0;j<N;j++) {
      if (A[i][j]>maximo) maximo = A[i][j];
    }
  }
```

```
  /* Una vez localizado el máximo, buscar sus posiciones */
  for (i=0;i<N;i++) {
    for (j=0;j<N;j++) {
      if (A[i][j] == maximo) {
        posiciones[k][0] = i;
        posiciones[k][1] = j;
        k = k+1;
      }
    }
  }
  return k;
}
```

(a) Paraleliza dicha función de forma eficiente mediante OpenMP, empleado una única región paralela.

(b) Modifica el código del apartado anterior para que cada hilo imprima por pantalla su identificador y la cantidad de valores máximos que ha encontrado y ha incorporado a la matriz `posiciones`.

Cuestión 3–13

Se dispone de una matriz `M` que almacena datos sobre las actuaciones de los `NJ` jugadores de un equipo de baloncesto en distintos partidos. Cada una de las `NA` filas de la matriz corresponde a la actuación de un jugador en un partido, almacenando, en sus 4 columnas, el dorsal del jugador (numeración consecutiva de 0 a `NJ`-1), el número de puntos anotados por el jugador en el partido, el número de rebotes conseguidos y el número de tapones logrados. La valoración individual de un jugador por cada partido se calcula de este modo:

$$\text{valoracion} = \text{puntos} + 1{,}5 * \text{rebotes} + 2 * \text{tapones}$$

Paraleliza, mediante OpenMP y con una única región paralela, la siguiente función encargada de obtener y mostrar por pantalla el jugador que más puntos ha anotado en un partido, además de calcular la valoración media de cada jugador del equipo.

```
void valoracion(int M[][4], double valoracion_media[NJ]) {
  int i,jugador,puntos,rebotes,tapones,max_pts=0,max_anot;
  double suma_valoracion[NJ];
  int num_partidos[NJ];
  ...
  for (i=0;i<NA;i++) {
    jugador = M[i][0];
    puntos  = M[i][1];
    rebotes = M[i][2];
    tapones = M[i][3];
    suma_valoracion[jugador] += puntos+1.5*rebotes+2*tapones;
    num_partidos[jugador]++;
    if (puntos>max_pts) {
      max_pts  = puntos;
      max_anot = jugador;
    }
  }
  printf("Maximo anotador %d (%d puntos)\n",max_anot,max_pts);
  for (i=0;i<NJ;i++) {
    if (num_partidos[i]==0)
      valoracion_media[i] = 0;
    else
      valoracion_media[i] = suma_valoracion[i]/num_partidos[i];
```

```
  }
  ...
}
```

Cuestión 3–14

Se está celebrando un concurso de fotografía en el que los jueces otorgan puntos a aquellas fotos que deseen.

Se dispone de una función que recibe los puntos otorgados en las múltiples valoraciones efectuadas por todos los jueces y un vector `totales` donde se acumularán estos puntos. Este vector `totales` ya viene inicializado a ceros.

La función calcula los puntos totales para cada foto, mostrando por pantalla las dos mayores puntuaciones otorgadas a una foto en las valoraciones. También calcula y muestra la puntuación final media de todas las fotos así como el número de fotos que pasan a la siguiente fase del concurso, que son las que reciben un mínimo de 20 puntos.

Cada valoración `k` otorga una puntuación de `puntos[k]` a la foto número `indice[k]`. Lógicamente, una misma foto puede recibir múltiples valoraciones.

Paraleliza esta función de forma eficiente con OpenMP usando una sola región paralela.

```
/* nf = número de fotos, nv = número de valoraciones */
void concurso(int nf, int totales[], int nv, int indice[], int puntos[])
{
  int k,i,p,t, pasan=0, max1=-1,max2=-1, total=0;
  for (k = 0; k < nv; k++) {
    i = indice[k]; p = puntos[k];
    totales[i] += p;
    if (p > max2)
      if (p > max1) { max2 = max1; max1 = p; } else max2 = p;
  }
  printf("Las dos puntuaciones más altas han sido %d y %d.\n",max1,max2);
  for (k = 0; k < nf; k++) {
    t = totales[k];
    if (t >= 20) pasan++;
    total += t;
  }
  printf("Puntuación media: %g. %d fotos pasan a la siguiente fase.\n",
   (float)total/nf, pasan);
}
```

Cuestión 3–15

La siguiente función procesa la facturación, a final del mes, de todas las canciones descargadas por un conjunto de usuarios de una tienda de música virtual. Para cada una de las n descargas realizadas, se almacena el identificador del usuario y el de la canción descargada, respectivamente en los vectores `usuarios` y `canciones`. Cada canción tiene un precio diferente, recogido en el vector `precios`. La función además muestra por pantalla el identificador de la canción que se ha descargado en más ocasiones. Los vectores `ndescargas` y `facturacion` estarán inicializados a 0 antes de invocar a la función.

```
void facturaciones(int n, int usuarios[], int canciones[], float precios[],
                   float facturacion[], int ndescargas[])
{
  int i,u,c,mejor_cancion=0;
  float p;
```

```
  for (i=0;i<n;i++) {
     u = usuarios[i];
     c = canciones[i];
     p = precios[c];
     facturacion[u] += p;
     ndescargas[c]++;
  }
  for (i=0;i<NC;i++) {
     if (ndescargas[i]>ndescargas[mejor_cancion])
        mejor_cancion = i;
  }
  printf("La canción %d es la más descargada\n",mejor_cancion);
}
```

(a) Paraleliza eficientemente la función anterior empleando una única región paralela.

(b) ¿Sería válido emplear la cláusula `nowait` en el primero de los bucles?

(c) Modifica el código de la función paralelizada de modo que cada hilo muestre por pantalla su identificador y el número de iteraciones del primer bucle que ha procesado.

Cuestión 3–16

Queremos obtener la distribución de las calificaciones obtenidas por los alumnos de CPA calculando el número de suspensos, aprobados, notables, sobresalientes y matrículas de honor.

```
void histograma(int histo[], float notas[], int n) {
  int i, nota;
  float rnota;
  for (i=0;i<5;i++) histo[i] = 0;
  for (i=0;i<n;i++) {
    rnota = round(notas[i]*10)/10.0;
    if (rnota<5) nota = 0;        /* suspenso */
    else
      if (rnota<7) nota = 1;      /* aprobado */
      else
        if (rnota<9) nota = 2;    /* notable */
        else
          if (rnota<10) nota = 3; /* sobresaliente */
          else
            nota = 4;             /* matricula de honor */
    histo[nota]++;
  }
}
```

(a) Paraleliza adecuadamente la función `histograma` con OpenMP.

(b) Modifica la función `histograma` para que muestre por pantalla el número del alumno con la mejor nota y su nota, y el valor de la peor nota (ambas sin redondear).

Cuestión 3–17

La siguiente función gestiona un número determinado de viajes, que han tenido lugar durante un periodo concreto de tiempo, mediante el servicio público de bicicletas de una ciudad. Para cada uno de los viajes realizados, se almacenan los identificadores de las estaciones origen y destino, junto con el tiempo (expresado en minutos) de duración de cada uno de ellos. El vector `num_bicis` guarda el número de bicicletas presentes en cada estación. Además, la función calcula entre qué estaciones tuvo lugar el viaje más largo y el más corto, junto con el tiempo medio de duración de la totalidad de los viajes.

```
struct viaje {
   int estacion_origen;
   int estacion_destino;
   float tiempo_minutos;
};
void actualiza_bicis(struct viaje viajes[],int num_viajes,int num_bicis[]) {
   int i,origen,destino,ormax,ormin,destmax,destmin;
   float tiempo,tmax=0,tmin=9999999,tmedio=0;
   for (i=0;i<num_viajes;i++) {
      origen  = viajes[i].estacion_origen;
      destino = viajes[i].estacion_destino;
      tiempo  = viajes[i].tiempo_minutos;
      num_bicis[origen]--;
      num_bicis[destino]++;
      tmedio += tiempo;
      if (tiempo>tmax) {
         tmax=tiempo; ormax=origen; destmax=destino;
      }
      if (tiempo<tmin) {
         tmin=tiempo; ormin=origen; destmin=destino;
      }
   }
   tmedio /= num_viajes;
   printf("Tiempo medio entre viajes: %.2f minutos\n",tmedio);
   printf("Viaje más largo (%.2f min.) estación %d a %d\n",tmax,ormax,destmax);
   printf("Viaje más corto (%.2f min.) estación %d a %d\n",tmin,ormin,destmin);
}
```

Paraleliza la función mediante OpenMP de la forma más eficiente posible.