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];
}
```

Solución: Hay una dependencia de datos entre las diferentes iteraciones: incumple la 1ª condición de Bernstein ($I_j \cap O_i \neq \emptyset$), pues, por ejemplo, x[2] es una variable de salida en la iteración i=1 y una variable de entrada en la iteración i=2. No es posible eliminar esa dependencia de datos.

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

Solución: Hay una dependencia de datos entre las diferentes iteraciones: incumple la 3ª condición de Bernstein ($O_i \cap O_j \neq \emptyset$), pues `x` es una variable de salida en todas las iteraciones. En este caso sí es posible eliminar la dependencia:

```
for (i=0;i<N;i++) {
  a[i] = a[i] + y[i];
}
x = a[N-1];
```

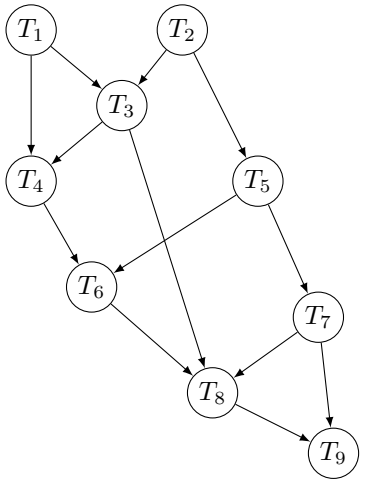
(c)
```
for (i=N-2;i>=0;i--) {
  x[i] = x[i] + y[i+1];
  y[i] = y[i] + z[i];
}
```

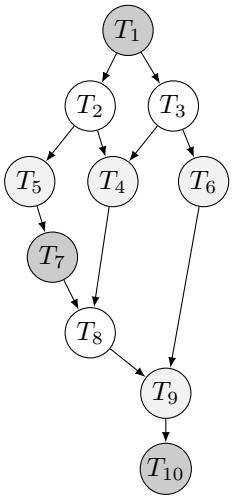
Solución: Hay una dependencia de datos entre las diferentes iteraciones: incumple la 1ª condición de Bernstein ($I_j \cap O_i \neq \emptyset$), pues, por ejemplo, y[1] es una variable de salida en la iteración i=1 y de entrada en la iteración i=0. En este caso sí es posible eliminar la dependencia:

```
x[N-2] = x[N-2] + y[N-1];
for (i=N-3;i>=0;i--) {
  y[i+1] = y[i+1] + z[i+1];
  x[i] = x[i] + y[i+1];
}
y[0] = y[0] + z[0];
```

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.

Solución: De entre todos los posibles caminos entre un nodo inicial y un nodo final, el que mayor coste tiene (camino crítico) es $T_1 - T_3 - T_4 - T_6 - T_8 - T_9$ (o de forma equivalente empezando en T_2). Su longitud es $L = 6$. El grado medio de concurrencia es

$$M = \sum_{i=1}^{9} \frac{1}{6} = \frac{9}{6} = 1{,}5$$

(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.

Solución: En este caso, el camino crítico es $T_1 - T_2 - T_5 - T_7 - T_8 - T_9 - T_{10}$ y su longitud es

$$L = \frac{1}{3}n^3 + n^3 + 2n^3 + \frac{1}{3}n^3 + n^3 + 2n^3 + \frac{1}{3}n^3 = 7n^3 \text{ flops}$$

El grado medio de concurrencia es

$$M = \frac{3 \cdot \frac{1}{3}n^3 + 3 \cdot n^3 + 4 \cdot 2n^3}{7n^3} = \frac{12n^3}{7n^3} = 1{,}71$$

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.

Solución:

(a)
```
void prodmvp(double a[N], double c[N], double B[N][N])
{
  int i, j;
  double sum;
  #pragma omp parallel for private(j,sum)
  for (i=0; i<N; i++) {
    sum = 0.0;
    for (j=0; j<N; j++)
      sum += B[i][j] * c[j];
    a[i] = sum;
  }
}
```

(b) Coste secuencial: $t(N) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} 2 = 2N^2$ flops

Coste paralelo: $t(N,p) = \sum_{i=0}^{d-1} \sum_{j=0}^{N-1} 2 = 2dN$ flops, donde $d = \frac{N}{p}$

(c) Speedup: $S(N,p) = \frac{t(N)}{t(N,p)} = \frac{2N^2}{2dN} = \frac{N}{d} = p$

Eficiencia: $E(N,p) = \frac{S(N,p)}{p} = \frac{p}{p} = 1$

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).

Solución: El cálculo tiene dos fases. En la primera se sobrescribe cada fila de la matriz con la fila siguiente multiplicada por 2. En la segunda parte se realiza la suma de todos los elementos de la matriz.

En la primera fase se realiza sólo una operación en el bucle más interior, y por tanto su coste es $\sum_{i=0}^{M-2}\sum_{j=0}^{N-1} 1 = \sum_{i=0}^{M-2} N = (M-1)N \approx MN$. [La aproximación la hacemos en sentido asintótico, es decir, suponiendo que tanto N como M son suficientemente grandes.] La segunda fase tiene un coste similar, salvo que el bucle `i` hace una iteración más: MN.

Coste secuencial: $t_1 = 2MN$ flops

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

Solución: Planteamos una paralelización con dos regiones paralelas, una por cada fase, ya que la segunda fase no puede empezar hasta que haya terminado la primera. En la primera fase existen dependencias de datos en el índice `i`, que pueden resolverse intercambiando los bucles y paralelizando el bucle `j` (también se podría paralelizar el bucle `j` sin intercambiar los bucles, pero esto sería más ineficiente). La segunda fase requiere una reducción sobre la variable `suma`. En ambas fases tanto `i` como `j` deben ser variables privadas.

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

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

Solución: En paralelo, el coste de la primera fase sería $\frac{N}{p}M$ y el de la segunda fase $\frac{M}{p}N$ (en este segundo término hemos despreciado el coste de la reducción que hace internamente OpenMP sobre la variable `suma`).

El tiempo paralelo sería: $t_p = \dfrac{2MN}{p}$ flops

El speedup será por tanto: $S_p = \dfrac{t_1}{t_p} = \dfrac{2MN}{2MN/p} = p$

(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).

Solución: En este caso, $t_p = MN + \dfrac{MN}{p}$ flops, y por tanto

$$S_p = \frac{2MN}{MN + MN/p} = \frac{2}{1 + 1/p} = \frac{2p}{p+1},$$

cuyo límite cuando $p \to \infty$ es 2. Es decir, el speedup nunca podrá ser mayor que dos aunque usemos muchos procesadores.

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?

Solución: Primer bucle:

```
#pragma omp parallel for reduction(+:s) private(j,k,aux)
```

Segundo bucle:

```
#pragma omp parallel for reduction(+:s) private(k,aux)
```

Tercer bucle:

```
#pragma omp parallel for reduction(+:aux)
```

La forma más eficiente consiste en paralelizar el bucle más externo, pues se produce una menor sobrecarga debida a la activación y desactivación de hilos, y también se reducen los tiempos de espera debidos a la sincronización implícita al final de la directiva.

(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.

Solución: Coste secuencial: $t_1 = \displaystyle\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\left(1 + \sum_{k=0}^{n-1} 2\right) \approx 2n^3$ flops.

Coste paralelo: $t_n = 2n^3/n = 2n^2$ flops.
Speedup: $S_n = t_1/t_n = n$.

(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.

Solución: Se declararían dos variables enteras, `iter` y `tid` (`iter` debe estar inicializada a 0 y `tid` debe aparecer como privada en la cláusula `parallel`, con `private(tid)`), y se añadirían las siguientes líneas:
Entre el primer y el segundo `for`:

```
tid = omp_get_thread_num();
if (!tid) iter++;
```

Después de los bucles anidados (antes del `return`):

```
printf("Iteraciones realizadas por el hilo 0 = %d\n",iter);
```

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.

Solución:

```
#include <stdio.h>
#include <omp.h>

int main() {
  int i, tid;
  unsigned int n, s=0;
  scanf("%u",&n);
  omp_set_num_threads(6);
  #pragma omp parallel for lastprivate(tid) reduction(+:s) schedule(dynamic,2)
  for (i=1;i<=n;i++) {
    tid = omp_get_thread_num();
    s+=i;
  }
  printf("El hilo que ha sumado el último número es %d\n",tid);
  printf("La suma de los primeros %u números naturales es %u\n",n,s);
}
```

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.

Solución:

```
#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;
     #pragma omp parallel for private(j) reduction(+:err)
     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;
  }
```

(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.

Solución:

$$t(n) = \sum_{i=0}^{n-1}(\sum_{j=0}^{i-1} 2 + \sum_{j=i+1}^{n-1} 2 + 3) = \sum_{i=0}^{n-1}(2n+1) \approx 2n^2$$

$$t(n,p) = \sum_{i=0}^{\frac{n}{p}-1}(\sum_{j=0}^{i-1} 2 + \sum_{j=i+1}^{n-1} 2 + 3) = \sum_{i=0}^{\frac{n}{p}-1}(2n+1) \approx \frac{2n^2}{p}$$

$$S(n,p) = \frac{2n^2}{\frac{2n^2}{p}} = p$$

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.

Solución: El bucle más exterior no se puede paralelizar debido a una dependencia de cada iteración con la anterior, al utilizar como `x` el valor obtenido en `x2` en la iteración anterior. En el bucle `i` la variable `s` acumula el valor del producto de la fila por el vector pero se calcula de forma completa en cada iteración, por lo que deberá ser privada (al igual que la variable del bucle más interior,

j). No es éste el caso de la variable q, cuyo valor es el resultado de multiplicar los valores de las diferentes iteraciones, requiriendo una reducción. Al haber una dependencia entre los dos bucles for de i, no es necesario utilizar una única región paralela ya que no podemos utilizar la cláusula nowait. La variable max no necesita protección porque no hay condiciones de carrera.

```
#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;
    #pragma omp parallel for private(s,j) reduction(*:q)
    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;
    }

    #pragma omp parallel for
    for (i=0;i<n;i++)
      x[i] = x2[i];

    if (max<q)
      max = q;
  }
  return max;
}
```

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

Solución:

$$t(n) = \sum_{i=0}^{n-1} \big(\sum_{j=0}^{n-1} 2 + 1 \big) = \sum_{i=0}^{n-1} (2n+1) = 2n^2 + n \approx 2n^2 \text{ flops}$$

(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.

Solución:

$$t(n,p) = \sum_{i=0}^{\frac{n}{p}-1} \big(\sum_{j=0}^{n-1} 2 + 1 \big) = 2\frac{n^2}{p} + \frac{n}{p} \approx 2\frac{n^2}{p}$$

$$S(n,p) = \frac{2n^2}{2\frac{n^2}{p}} = p$$

$$E(n,p) = 1$$

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`.

Solución: Bastaría con añadir la siguiente directiva inmediatamente antes del bucle:

```
#pragma omp parallel for private(j,k,mf,val)
```

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

Solución: Bastaría con añadir la siguiente directiva inmediatamente antes del bucle:

```
#pragma omp parallel for private(k,val) reduction(min:mf)
```

(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.

Solución: El tiempo de una iteración del bucle i sería:

$$1 + \sum_{j=0}^{N-1} \left(2 + \sum_{k=0}^{i-1} 2 \right) \approxeq \sum_{j=0}^{N-1} 2i = 2Ni \,\text{flops}$$

y el tiempo de la función completa es la suma del tiempo de todas las iteraciones del bucle, o sea:

$$\sum_{i=0}^{M-1} 2Ni = 2N \sum_{i=0}^{M-1} i \approxeq NM^2 \,\text{flops}$$

(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.

Solución: En el apartado anterior vemos que el coste de una iteración del bucle `i` es mayor cuanto mayor sea el valor de `i`. Si usamos la planificación `schedule(static)` el equilibrio de carga será malo, puesto que el bloque de iteraciones más costosas (con mayor valor de `i`) le tocarán a un mismo hilo.

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.

Solución: Es conveniente paralelizar el bucle más externo. Bastaría con añadir la siguiente directiva inmediatamente antes del bucle `i`:

```
#pragma omp parallel for private(j, k, aux) reduction(+: s)
```

Las distintas iteraciones del bucle `i` tienen coste diferente, pues el recorrido del bucle `k` depende del valor de `i`. Por tanto, es de esperar que la planificación sí afecte. Como para dos valores consecutivos del valor `i` el valor de `k` varia en 1, podrían ser adecuadas las planificaciones `static` o `dynamic` con el valor de *chunk* igual a 1, es decir, `schedule(static,1)` o `schedule(dynamic,1)`.

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

Solución:

$$t(N) = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\left(2+\sum_{k=i}^{N-1}2\right) \cong 2\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}(N-i) = 2\sum_{i=0}^{N-1}\left(N^2-iN\right)$$
$$\cong 2\left(N^3-\frac{N^3}{2}\right) \cong N^3 \text{ flops}$$

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;
}
```

Solución: Declaramos como `volatile` la variable `encontrado` para garantizar que en el momento en que un hilo modifique su valor el resto de hilos verán dicho cambio.

```
int busqueda(int x[], int n, int valor)
{
   volatile int encontrado=0;
   int i, salto;
   #pragma omp parallel private(i)
   {
      i = omp_get_thread_num();
      salto = omp_get_num_threads();
      while (!encontrado && i<n) {
         if (x[i]==valor) encontrado=1;
         i += salto;
      }
   }
   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`.

Solución:

```
double norma(double v[], int n)
{
   int i, i_hilo, p;
```

```
   double r=0;

   /* Fase 1 */
   #pragma omp parallel private(i_hilo)
   {
      p = omp_get_num_threads();
      i_hilo = omp_get_thread_num();
      sumas[i_hilo]=0;
      #pragma omp for schedule(static)
      for (i=0; i<n; i++)
         sumas[i_hilo] += v[i]*v[i];
   }

   /* Fase 2 */
   for (i=0; i<p; i++)
      r += sumas[i];
   return sqrt(r);
}
```

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

Solución:

```
double norma(double v[], int n)
{
   int i;
   double r=0;
   #pragma omp parallel for reduction(+:r)
   for (i=0; i<n; i++)
      r += v[i]*v[i];
   return sqrt(r);
}
```

(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.

Solución: Coste del algoritmo secuencial (el coste de la raíz cuadrada es despreciable frente al coste del bucle `i`):

$$t(n) = \sum_{i=0}^{n-1} 2 \approx 2n \text{ flops}$$

Coste del algoritmo paralelo: como cada iteración del bucle `i` cuesta 2 flops y cada hilo realiza n/p iteraciones, el coste es de $t(n,p) = 2n/p$ flops. El speedup es de $S(n,p) = 2n/(2n/p) = p$.

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.

Solución:

```
void f(int n, double a[], double b[])
{
   int i, cont;
   #pragma omp parallel private(cont)
   {
      cont=0;
      #pragma omp for
      for (i=0; i<n; i++) {
         b[i]=cos(a[i]);
         cont++;
      }
      printf("Hilo %d: %d iteraciones procesadas\n",
                omp_get_thread_num(), cont);
   }
}
```

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.

Solución: El cálculo tiene dos fases. En la primera se calcula la suma de los cuadrados de los elementos de la matriz, en la segunda se escalan los elementos de la matriz. Con dos regiones paralelas se garantiza que la segunda fase no empieza antes de que haya terminado la primera, de lo contrario el cálculo podría ser incorrecto. La primera fase requiere una reducción sobre la variable `suma`. En ambos casos la variable `j` debe ser privada. La variable `factor` es compartida y la calcula el hilo principal.

```
void normaliza(double A[N][N])
{
  int i,j;
  double suma=0.0,factor;
  #pragma omp parallel for reduction(+:suma) private(j)
  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);
  #pragma omp parallel for private(j)
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      A[i][j] = factor*A[i][j];
    }
  }
}
```

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

Solución:

```
void normaliza(double A[N][N])
{
  int i,j;
  double suma=0.0,factor;
  #pragma omp parallel private(j)
  {
    #pragma omp for reduction(+:suma)
    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);
    #pragma omp for
    for (i=0; i<N; i++) {
      for (j=0; j<N; j++) {
        A[i][j] = factor*A[i][j];
      }
    }
  }
}
```

La cláusula `nowait` se utilizaría para eliminar la barrera implícita al final de la primera directiva `for`. Sin embargo, en este caso la barrera es necesaria, ya que sería incorrecto que uno de los hilos pasara a ejecutar el siguiente `for` mientras otros hilos están aún en el anterior (la segunda fase modifica la matriz mientras que la primera fase la lee). Por otro lado, la cláusula `nowait` en combinación con `reduction` podría hacer que alguno de los hilos utilizara un valor incorrecto de `factor`, antes de completarse la reducción.

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.

Solución:

```
double ejp(double x[M], double y[N], double A[M][N])
{
   int i, j;
   double aux,s=0.0;
   #pragma omp parallel
   {
      #pragma omp for nowait
      for (i=0; i<M; i++)
         x[i] = x[i]*x[i];
      #pragma omp for
      for (i=0; i<N; i++)
         y[i] = 1.0+y[i];
      #pragma omp for private(j,aux) reduction(+:s)
      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;
}
```

Como entre los dos primeros bucles `for` no hay dependencia de datos, es conveniente utilizar la cláusula `nowait` en el primer bucle paralelo `for`, para así evitar la sincronización que se produce en la finalización de la directiva `for`.

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

Solución:
Tiempo secuencial:

$$t(M,N) = M + N + 2MN \approx 2MN \text{ flops},$$

suponiendo M y N suficientemente grandes (coste asintótico).
Tiempo paralelo:

$$t(M,N,p) = \frac{M}{p} + \frac{N}{p} + \frac{2MN}{p} = \frac{M+N+2MN}{p} \approx \frac{2MN}{p} \text{ flops},$$

suponiendo M y N suficientemente grandes (coste asintótico).

(c) Determina el speedup y la eficiencia.

Solución: Speedup:

$$S(M,N,p) = \frac{t(M,N)}{t(M,N,p)} \approx \frac{2MN}{\frac{2MN}{p}} = p$$

Eficiencia:

$$E(M,N,p) = \frac{S(M,N,p)}{p} = 1$$

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;
```

Solución: La única cosa a tener en cuenta es que la variable `a` debe ser privada.

```
#pragma omp parallel sections private(a)
{
  #pragma omp section
  {
    a = -1.8;
    fun1(n,&a);
    b[0] = a;
  }
  #pragma omp section
  {
    a = 3.2;
```

```
      fun2(n,&a);
      b[1] = a;
    }
    #pragma omp section
    {
      a = 0.25;
      fun3(n,&a);
      b[2] = a;
    }
  }
```

Cuestión 2–7

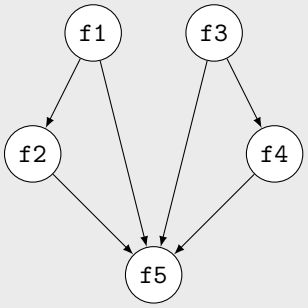
Dada la siguiente función:

```
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.

Solución: El grafo de dependencias se muestra a continuación:



Las dependencias de `f5` con respecto de las otras tareas son dependencias de flujo (sus entradas son generadas por otras tareas). Las dependencias de `f2` con `f1` y `f4` con `f3` son anti-dependencias (no es que necesiten la salida de otras tareas, sino que modifican la entrada de tareas previas y por tanto no deben realizarse hasta que no han terminado esas tareas previas). También existe una dependencia de salida entre `f5` y `f1` ya que ambas modifican el mismo dato de salida (la `a`).

(b) Paraleliza la función mediante directivas OpenMP.

Solución:

```
void func(double a[],double b[],double c[],double d[])
{
  #pragma omp parallel sections
  {
```

```
        #pragma omp section
        {
          f1(a,b);
          f2(b,b);
        }
        #pragma omp section
        {
          f3(c,d);
          f4(d,d);
        }
      }
      f5(a,a,b,c,d);
    }
```

(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?

Solución: Por comodidad, asumimos que cada función tarda 1 unidad de tiempo. El tiempo secuencial es

$$t_1 = 1 + 1 + 1 + 1 + 1 = 5$$

Para conseguir un speedup grande conviene reducir lo máximo posible el tiempo de ejecución. Esto nos lleva a usar tantos procesadores como tareas paralelas se pueden ejecutar concurrentemente. Dado el programa paralelo y el grafo de dependencias, basta con 2 procesadores (hilos). En ese caso, el coste sería (`f1` y `f2` se hacen en paralelo a `f3` y `f4`):

$$t_p = 2 + 1 = 3$$

$$Sp = t_1/t_p = 5/3 = 1{,}67$$

Para mejorar el speedup, se podrían eliminar las anti-dependencias replicando los datos que van a ser modificados. Habría que ver si el coste de la copia y el espacio extra necesario compensan la ganancia de velocidad que se obtendría. Haciendo esto, se copiarían `b` y `d` antes de empezar a trabajar. `f1` y `f3` trabajarían con las copias y `f2` y `f4` con los datos reales. Con ello se podrían hacer a la vez `f1`,`f2`,`f3`,`f4`, con lo que se tendría (sin tener en cuenta el sobrecoste de las copias):

$$t_p = 1 + 1 = 2 \qquad \text{(trabajando con 4 procesadores)}$$

$$S_p = 5/2 = 2{,}5$$

Cuestión 2–8

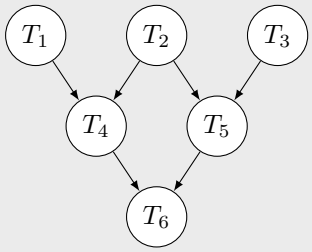
En la siguiente función, `T1`, `T2`, `T3` modifican `x`, `y`, `z`, respectivamente.

```
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.

Solución:



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

Solución:

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

   #pragma omp parallel private(i,j)
   {
      #pragma omp sections
      {
         #pragma omp section
         T1(x,n);    /* Tarea T1 */
         #pragma omp section
         T2(y,n);    /* Tarea T2 */
         #pragma omp section
         T3(z,n);    /* Tarea T3 */
      }
      #pragma omp sections
      {
         #pragma omp section
         /* 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;
            }
            #pragma omp section
            /* 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;
   }
```

(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.

Solución: El coste a priori de T_4 es de:

$$\sum_{i=0}^{n-1}\left(\sum_{j=0}^{n-1}2+\sum_{j=0}^{n-1}1\right)=\sum_{i=0}^{n-1}(2n+n)=3n^2 \text{ flops}$$

El coste de T_5 es igual al de T_4, y el de T_6 es de $2n+1$ flops.
Por tanto, el coste secuencial es de:

$$t(n)=2n^2+2n^2+2n^2+3n^2+3n^2+2n+1\approx 12n^2 \text{ flops}$$

Si el número de hilos, p, es al menos 3, el coste del algoritmo paralelo será de $t(n,p)=2n^2+3n^2+2n\approx 5n^2$ flops, y el speedup será:

$$S(n,p)=\frac{12n^2}{5n^2}=2{,}4$$

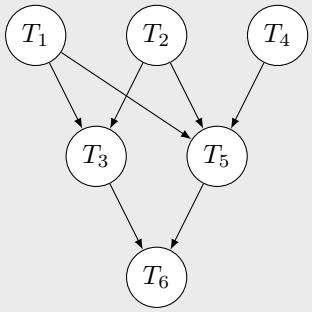
Cuestión 2–9

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.

Solución: La tarea T_5 modifica `x`, por lo que tiene una anti-dependencia con respecto de T_1, T_2 y T_4.



(b) Paraleliza el código mediante OpenMP.

Solución:

```
#pragma omp parallel
{
   #pragma omp sections
   {
      #pragma omp section
      minx = minimo(x,n);       /* T1 */
      #pragma omp section
      maxx = maximo(x,n);       /* T2 */
      #pragma omp section
      calcula_y(y,x,n);         /* T4 */
   }
   #pragma omp sections
   {
      #pragma omp section
      calcula_z(z,minx,maxx,n); /* T3 */
      #pragma omp section
      calcula_x(x,y,n);         /* T5 */
   }

}
calcula_v(v,z,x);          /* T6 */
```

(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.

Solución: Longitud del camino crítico: $L = 2n + n + n = 4n$ flops
Grado medio de concurrencia: $\frac{7n}{4n} = 7/4 = 1{,}75$
Para obtener el speedup y la eficiencia, hay que tener en cuenta que el tiempo de ejecución secuencial es $t(n) = 7n$, y el tiempo de ejecución paralelo con 5 procesadores es $t(n, 5) = 2n + n + n = 4n$ flops. Por tanto:

$$S(n,p) = \frac{7n}{4n} = 1{,}75$$

$$E(n,p) = \frac{1{,}75}{5} = 0{,}35$$

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);
```

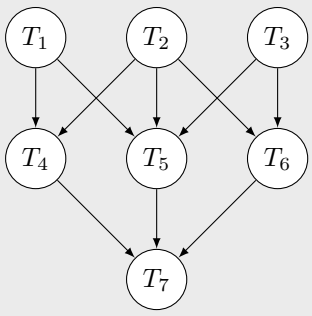
(a) Paraleliza el código de forma eficiente a nivel de bucles.

Solución: El bucle de `fun1` no se puede paralelizar debido a las dependencias entre las distintas iteraciones, por lo que se considera únicamente la función `compara`.

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

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

Solución:



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

Solución: El código paralelo del programa principal es el siguiente (el resto queda invariante):

```
int i, n=10;
double a[10], b[10], c[10], x=5, y=7, z=11, w;

#pragma omp parallel sections
{
  #pragma omp section
  fun1(a,n,x);
  #pragma omp section
  fun1(b,n,y);
  #pragma omp section
  fun1(c,n,z);
}
#pragma omp parallel sections
{
  #pragma omp section
  x = compara(a,b,n);
  #pragma omp section
  y = compara(a,c,n);
  #pragma omp section
  z = compara(c,b,n);
}
w = x+y+z;
printf("w:%f\n", w);
```

(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.

Solución: El tiempo secuencial es:

$$t(n) = 3\sum_{i=1}^{n-1} 1 + 3\sum_{i=0}^{n-1} 3 + 2 \approx 3n + 9n = 12n$$

El tiempo paralelo y el speed-up para el primer algoritmo considerando 3 procesadores es:

$$t_1(n,3) = 3\sum_{i=1}^{n-1} 1 + 3\sum_{i=0}^{\frac{n}{3}-1} 3 + 2 \approx 3n + 3n = 6n$$

$$S_1(n,3) = \frac{12n}{6n} = 2$$

El tiempo paralelo y el speed-up para el segundo algoritmo considerando 3 procesadores es:

$$t_2(n,3) = \sum_{i=1}^{n-1} 1 + \sum_{i=0}^{n-1} 3 + 2 \approx n + 3n = 4n$$

$$S_2(n,3) = \frac{12n}{4n} = 3$$

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);
```

Solución: La paralelización a nivel de bucles no es posible en este caso ya que el número de iteraciones no es conocido a priori. Por tanto, se aborda una paralelización por tareas, ubicando cada bucle (incluyendo la inicialización) en una sección concurrente. Para que el resultado sea correcto, deberemos además tener en cuenta que las variables `x`, `y`, `z` deben tener un ámbito privado, ya que las modifican los tres bucles pero su valor no es compartido entre los bucles (se inicializan al principio de cada bucle). Finalmente, la variable `w` debe tener como valor final el valor acumulado de todas las secciones concurrentes, para lo que se utilizará una cláusula `reduction`.

```
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 */

#pragma omp parallel sections private(x,y,z) reduction(+:w)
{
  #pragma omp section
  {
    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);
  }
  #pragma omp section
  {
    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);
  }
  #pragma omp section
  {
    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.

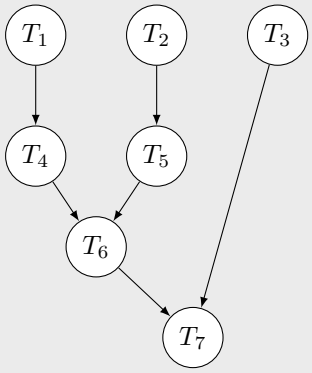
Solución: En ambas funciones, la forma más sencilla y eficiente de realizar la paralelización a nivel de bucles es paralelizar mediante la directiva `parallel for` el bucle más externo. De este modo, las iteraciones de dicho bucle se reparten entre los diferentes hilos de manera que cada hilo se encarga del cálculo asociado a un conjunto determinado de filas de la matriz resultado. Debemos definir, adicionalmente, el alcance de las variables, de modo que las variables `j`, `k` y `suma` deben ser privadas para cada hilo.

```
void matmult(double A[N][N],
       double B[N][N],double C[N][N])
{
  int i,j,k;
  double suma;
  #pragma omp parallel for private(j,k,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;
    }
  }
}
```

```
void simetriza(double A[N][N])
{
  int i,j;
  double suma;
  #pragma omp parallel for private(j,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;
    }
  }
}
```

(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.

Solución: Las 3 primeras tareas no presentan dependencias entre sí, por lo que pueden ejecutarse concurrentemente. Existe por el contrario una dependencia entre las tareas T_1 y T_4, y entre las tareas T_5 y T_2 (debida a las variables `C1` y `C2`, respectivamente). Las tareas T_4 y T_5 también pueden ejecutarse simultáneamente al no existir dependencia alguna entre ellas. Los datos de entrada de la tarea T_6 (`C1` y `C2`) son los datos de salida de T_4 y T_5, existiendo por tanto una dependencia de flujo entre ellas, y de forma similar entre T_3 y T_6 y la tarea T_7. El grafo de dependencias es el siguiente:



El grado máximo de concurrencia es 3, ya que como mucho habrá 3 tareas ejecutándose concurrentemente.
El coste en flops de `matmult` (c_m) y de `simetriza` (c_s) es:

$$c_m = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\sum_{k=0}^{N-1} 2 = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} 2N = \sum_{i=0}^{N-1} 2N^2 = 2N^3.$$

$$c_s = \sum_{i=0}^{N-1}\sum_{j=0}^{i} 1 = \sum_{i=0}^{N-1}(i+1) \approx \frac{N^2}{2}.$$

Por tanto, cinco de las tareas tienen un coste de $2N^3$, mientras que T_4 y T_5 tienen un coste mucho menor ($N^2/2$). El coste total acumulado de todas las tareas es $C = 10N^3 + N^2$ (coste secuencial).

El camino crítico es T_1–T_4–T_6–T_7 (o, equivalentemente, T_2–T_5–T_6–T_7), cuyo coste es

$$L = 2N^3 + \frac{N^2}{2} + 2N^3 + 2N^3 = 6N^3 + \frac{N^2}{2}.$$

El grado medio de concurrencia es

$$M = \frac{C}{L} = \frac{10N^3 + N^2}{6N^3 + N^2/2} \approx \frac{10}{6} = 1,67.$$

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

Solución: Podemos agrupar las tareas T_1 con T_4 y T_2 con T_5, ya que no existen dependencias cruzadas. Entonces, una posible solución sería hacer una primera parte con dos secciones, y posteriormente ejecutar T_3 y T_6 en paralelo. Finalmente quedaría ejecutar la tarea T_7 tras la finalización de las anteriores. Esta aproximación sólo requiere dos hilos concurrentes. También podrían ejecutarse los dos primeros grupos con T_3 en paralelo y ejecutar secuencialmente T_6 y T_7. Incluimos la primera solución. Para ello, dentro de una única región paralela usamos dos veces la directiva `sections` (con dos secciones concurrentes cada vez) para implementar mediante barreras implícitas las dependencias de la tarea T_6 con las tareas T_4 y T_5. Con dicha implementación, todas las variables serán compartidas, puesto que no existirán tareas que se estén ejecutando simultáneamente y que modifiquen las mismas variables.

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    {
      matmult(X,Y,C1);
      simetriza(C1);
    }
    #pragma omp section
    {
      matmult(Y,Z,C2);
      simetriza(C2);
    }
  }
  #pragma omp sections
  {
    #pragma omp section
    matmult(Z,X,C3);
    #pragma omp section
    matmult(C1,C2,D1);
  }
}
matmult(D1,C3,D);
```

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.

Solución:

$$t_1 = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} 1 + \sum_{i=1}^{N-1} 1 + \sum_{j=0}^{N-1}\sum_{i=0}^{N-1} 1 = N^2 + N - 1 + N^2 = 2N^2 + N - 1 \approx 2N^2$$

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

Solución: El bucle que realiza la suma prefija no se puede paralelizar, dado que existen dependencias de datos entre las distintas iteraciones. Existen esquemas algorítmicos (bastante complicados) para calcular la suma prefija en paralelo. Sin embargo, en esta ocasión hemos decidido realizar esa fase de forma secuencial, ya que tiene un coste lineal, mientras que las otras dos partes tienen coste cuadrático, por lo que la penalización por no paralelizar esa operación es pequeña.

Para hacer la suma prefija de forma secuencial es necesaria una directiva `single` porque si todos los hilos ejecutaran esa parte habría condiciones de carrera.

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

```
          }
        }
      }
    }
```

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

Solución: El tiempo paralelo sería:

$$t_p = \sum_{i=0}^{N/p-1} \sum_{j=0}^{N-1} 1 + \sum_{i=1}^{N-1} 1 + \sum_{j=0}^{N/p-1} \sum_{i=0}^{N-1} 1 = \frac{N^2}{p} + N - 1 + \frac{N^2}{p} = \frac{2N^2}{p} + N - 1 \approx \frac{2N^2}{p}$$

El speedup será por tanto:

$$S_p = \frac{t_1}{t_p} \approx \frac{2N^2}{2N^2/p} = p$$

Se obtiene el speedup ideal, aunque el valor obtenido hubiera sido menor si en las expresiones de t_1 y t_p se hubieran mantenido los términos de orden lineal.

Cuestión 2–14

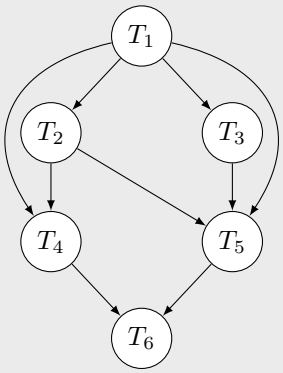
Dada la siguiente función:

```
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.

Solución: La tarea T_1 modifica sus dos argumentos, por lo que las tareas T_2 y T_3 tienen una dependencia con la tarea T_1. La tarea T_4 tiene una dependencia con T_1 y T_2, mientras que la tarea T_5 depende de T_1, T_2 y T_3. La tarea T_6 depende de T_4 y T_5. El grafo de dependencias sería así:



El grado máximo de concurrencia es 2 (por ejemplo, T_2 y T_3 se pueden hacer a la vez).

Teniendo en cuenta el coste de cada tarea, el camino crítico puede pasar indistintamente por T_2 o T_3 (tienen igual coste), pero tiene que pasar por T_5 puesto que T_5 tiene mayor coste que T_4. Hay dos posibles caminos críticos: el T_1, T_2, T_5, T_6 y el T_1, T_3, T_5, T_6. La longitud de cualquiera de los dos es:

$$L = 2n^2 + n^2 + 2n^2 + 1 = 5n^2 + 1 \quad \text{flops}$$

El grado medio de concurrencia sería:

$$M = \frac{\sum_{i=1}^{6} c_i}{L} = \frac{7n^2 + 1}{5n^2 + 1} \approx 1{,}4$$

(b) Paraleliza la función con OpenMP.

Solución: Utilizamos la directiva `sections` al tratarse de una aproximación basada en paralelismo de tareas. La dependencia existente entre las tareas T_2 y T_5 hace que no se puedan agrupar las tareas $T_2 - T_4$ y las tareas $T_3 - T_5$, siendo necesario que se produzca una sincronización entre el primer bloque de tareas concurrentes (T_2 y T_3) y el segundo (T_4 y T_5). Utilizaremos por tanto dos regiones paralelas con `sections` separadas. La tarea T_6 se realizará de forma secuencial al final de las regiones paralelas, al igual que T_1 al principio del programa.

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

  rellena(A,B);               /* T1 */
  #pragma omp parallel sections
  {
    #pragma omp section
    a = calculos(A);          /* T2 */
    #pragma omp section
    b = calculos(B);          /* T3 */
  }
  #pragma omp parallel sections
  {
    #pragma omp section
    x = suma_menores(B,a);    /* T4 */
    #pragma omp section
    y = suma_en_rango(B,a,b); /* T5 */
  }
```

```
    z = x + y;                  /* T6 */

    return z;
  }
```

(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.

Solución: Si se ejecuta con 2 o más hilos, como en cada `sections` del código hay un máximo de 2 `section`, ambas `section` podrán hacerse en paralelo. Por lo tanto, para el tiempo paralelo hay que coger el tiempo máximo de las diferentes tareas presentes en cada `sections`.

$$t_1 = 2n^2 + n^2 + n^2 + n^2 + 2n^2 + 1 = 7n^2 + 1 \quad \text{flops}$$

$$t_3 = 2n^2 + n^2 + 2n^2 + 1 = 5n^2 + 1 \quad \text{flops}$$

$$S_p = \frac{t_1}{t_3} \approx 1{,}4$$

$$E = \frac{Sp}{p} = \frac{Sp}{3} = 46{,}67\,\%$$

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).

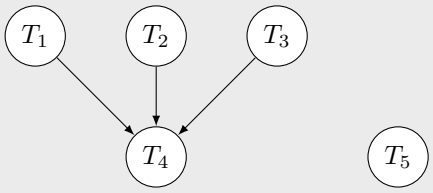
```
typedef double image[N][M];

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).

Solución: El grafo de dependencias se muestra a continuación:



El grado máximo de concurrencia sería 4, ya que T_1, T_2, T_3 y T_5 se pueden ejecutar en paralelo. Para obtener el grado medio de concurrencia, debemos determinar el coste en flops de las funciones `difer` y `suma`:

$$\text{Coste de } \texttt{difer} = \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} 1 = N \cdot M \qquad \text{Coste de } \texttt{suma} = \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} 2 = 2 \cdot N \cdot M$$

El camino crítico es $T_1 \longrightarrow T_4$ (o cualquiera de los otros dos caminos del grafo que terminan en T_4, que tienen el mismo coste), por lo que la longitud del camino crítico es:

$$L = N \cdot M + 2 \cdot N \cdot M = 3 \cdot N \cdot M$$

Por tanto,

$$\text{Grado medio de concurrencia} = \frac{N \cdot M + N \cdot M + N \cdot M + 2 \cdot N \cdot M + N \cdot M}{L} = \frac{6}{3} = 2$$

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

Solución: Realizamos la paralelización mediante secciones. La tarea T_5 se puede ejecutar en paralelo con cualquiera de las demás tareas. En el código paralelo la hemos situado en una construcción `sections` junto con T_4, ya que esta sería la solución con mayor concurrencia en el caso de tener 3 hilos.

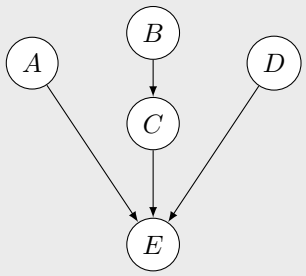
```
void procesa(image f1,image f2,image f3,image f4,image r1,image r2)
{
  image d1,d2,d3;
  #pragma omp parallel
  {
    #pragma omp sections
    {
      #pragma omp section
      difer(f2,f1,d1);          /* T1 */
      #pragma omp section
      difer(f3,f2,d2);          /* T2 */
      #pragma omp section
      difer(f4,f3,d3);          /* T3 */
    }
    #pragma omp sections
    {
      #pragma omp section
      suma(d1,d2,d3,r1);        /* T4 */
      #pragma omp section
      difer(f4,f1,r2);          /* T5 */
    }
  }
}
```

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.

Solución: Las tareas A, B y D no presentan ninguna dependencia entre sí y pueden ejecutarse concurrentemente. Por el contrario, la tarea C tiene, entre sus datos de entrada, a la variable `aux` generada por la tarea B. Existe por tanto una dependencia de flujo entre ellas. De igual modo, la tarea E tiene una dependencia de flujo con las tareas A, C y D, puesto que las 3 variables que suma se han obtenido a partir de la ejecución de las 3 tareas citadas.



Grado máximo de concurrencia: 3 (por ejemplo, se pueden hacer a la vez las tareas A, B y D).
El camino crítico es $A \rightarrow E$ y su longitud es $3n^2 + 2$ flops.
El grado medio de concurrencia es $\frac{3n^2+n^2+n^2+2n^2+2}{L} = \frac{7n^2+2}{3n^2+2} \approx 7/3 = 2{,}33$.

(b) Paralelízala con OpenMP.

Solución: Recurrimos a la directiva `sections` empleando una única región paralela y 3 secciones concurrentes, tras agrupar las tareas B y C (el coste agregado de estas tareas es menor o igual al de las otras tareas concurrentes A y D). La tarea E no forma parte de la región paralela y se ejecutará por el hilo principal cuando hayan acabado las tareas anteriores.

```
double calculos_matriciales(double mat[n][n])
{
  double x,y,z,aux,total;
  #pragma omp parallel sections
  {
    #pragma omp section
    x = A(mat);
    #pragma omp section
    {
      aux = B(mat);
      y = C(mat,aux);
    }
    #pragma omp section
```

```
      z = D(mat);
    }
    total = x + y + z;
    return total;
  }
```

(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.

Solución: $t_1 = 3n^2 + n^2 + n^2 + 2n^2 + 2 = 7n^2 + 2$ flops
Si ejecutamos el código paralelo con 2 hilos, el reparto de carga más equilibrado se corresponde con que un hilo ejecuta la tarea A y el otro ejecuta las tareas B, C y D. Finalmente, el hilo principal ejecuta la tarea E. De este modo,
$t_2 = \max(3n^2, n^2 + n^2 + 2n^2) + 2 = \max(3n^2, 4n^2) + 2 = 4n^2 + 2$ flops
$S_2 = \frac{t_1}{t_2} = \frac{7n^2+2}{4n^2+2} \approx 7/4 = 1{,}75$
$E_2 = \frac{S_2}{2} = \frac{1{,}75}{2} = 87{,}5\,\%$

(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.

Solución: Para calcular el tiempo de ejecución, empleamos la función `omp_get_wtime` antes y después de la región paralela correspondiente al apartado anterior, restando los valores de tiempo proporcionados para obtener el tiempo transcurrido. Fuera de la región paralela, el hilo principal mostrará dicho tiempo por pantalla. Para obtener el número de hilos, creamos una región paralela adicional donde invocamos la función `omp_get_num_threads`, mostrando el valor obtenido, una vez concluida la misma, por el hilo principal.

```
#include <omp.h>
double calculos_matriciales(double mat[n][n])
{
  double x,y,z,aux,total,t1,t2,t;
  int num_hilos;
  t1 = omp_get_wtime();
  #pragma omp parallel sections
  {
    #pragma omp section
    x = A(mat);
    #pragma omp section
    {
      aux = B(mat);
      y = C(mat,aux);
    }
    #pragma omp section
    z = D(mat);
  }
  total = x + y + z;
  t2 = omp_get_wtime();
  t = t2 - t1;
  #pragma omp parallel
  num_hilos = omp_get_num_threads();
  printf("Tiempo con %d hilos: %.2f segundos\n",num_hilos,t);
  return total;
```

```
    }
```

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.

Solución: Basta con colocar la siguiente directiva antes del bucle `i`, de modo que las variables `y`, `j`, `j2` y `a` sean privadas. Puesto que todos los hilos contribuyen parcialmente a la variable `x`, debemos emplear una reducción para acumular el valor obtenido por cada hilo.

```
#pragma omp parallel for private(y,j,j2,a) reduction(+:x)
```

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

Solución: El modo más eficiente consiste en emplear una única región paralela que abarque los dos bucles `j` y `j2`. De nuevo debemos emplear una reducción para obtener el valor correcto de las variables `x` e `y`, ya que todos los hilos participan en el cálculo de las mismas.

El uso de la cláusula `nowait` evita la barrera implícita entre los dos bucles, de modo que los hilos que ya han finalizado su labor en el bucle `j` pueden comenzar con el bucle `j2`. Esto es así puesto que no hay ninguna dependencia entre las variables de dichos bucles.

```
...
for (i=0; i<M; i++) {
   y = 1;
   #pragma omp parallel
   {
      #pragma omp for private(a) reduction(+:x) nowait
      for (j=0; j<N; j++) {
         ...
      }
      #pragma omp for reduction(*:y)
      for (j2=1; j2<i; j2++) {
```

```
            ...
        }
      }
      pf[i] = y;
    }
    ...
```

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

Solución:

$$\sum_{i=0}^{M-1}\left(\sum_{j=0}^{N-1}1+\sum_{j2=1}^{i-1}2\right)\approx\sum_{i=0}^{M-1}(N+2i)=\sum_{i=0}^{M-1}N+2\sum_{i=0}^{M-1}i\approx NM+M^2 \text{ flops}$$

(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.

Solución:

- Bucle `i`: Las distintas iteraciones del bucle `i` tienen coste diferente, porque el recorrido del bucle `j2` depende del valor de `i`. Por tanto, es de esperar que la planificación afecte. Serían adecuadas planificaciones *static* con *chunk* 1, *dynamic* o *guided*.
- Bucle `j`: todas las iteraciones del bucle tienen el mismo coste (1 flop), por lo que en principio la planificación no afectaría a las prestaciones (es posible que una planificación dinámica funcione peor por suponer cierta sobrecarga).
- Bucle `j2`: ocurre lo mismo que en el bucle `j` (en este caso cada iteración son 2 flops).

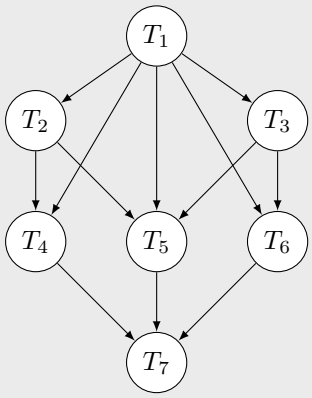
Cuestión 2–18

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.

Solución: A continuación se muestra el grafo de dependencias. Todas las dependencias son de flujo, excepto la antidependencia entre T_5 y las tareas T_2 y T_3 (T_5 modifica las variables `x` e `y`).



(b) Paraleliza la función eficientemente con OpenMP.

Solución: La función paralelizada sería la siguiente:

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

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

Solución: El tiempo secuencial es:

$$t(n) = 2n + 4n + 5n^2 = 5n^2 + 6n \simeq 5n^2$$

y el tiempo en paralelo con 3 procesadores:

$$t(n,p) = n + 2n + 3n^2 + n = 3n^2 + 4n \simeq 3n^2$$

A partir de dichos valores, el incremento de velocidad valdrá:

$$S(n,p) = \frac{t(n)}{t(n,p)} \simeq \frac{5n^2}{3n^2} = 1,67$$

y la eficiencia:

$$E(n,p) = \frac{S(n,p)}{p} \simeq \frac{1,67}{3} = 0,56$$

(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.

Solución: Uno de los caminos críticos es $T_1 - T_2 - T_5 - T_7$ y su longitud es:

$$L = n + 2n + 3n^2 + n = 3n^2 + 4n$$

A partir de la longitud del camino crítico, el grado medio de concurrencia es:

$$M = \frac{\sum_{i=1}^{7} C_i}{L} = \frac{2n + 4n + 5n^2}{3n^2 + 4n} = \frac{5n^2 + 6n}{3n^2 + 4n} = \frac{5n + 6}{3n + 4} \simeq \frac{5n}{3n} = 1,67$$

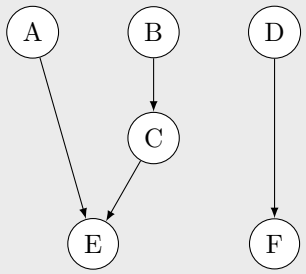
Cuestión 2–19

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.

Solución:



Camino crítico: A$\longrightarrow$E

Longitud del camino crítico: $L = 2n^2 + n^2 = 3n^2$ flops

Grado medio de concurrencia: $M = \frac{2n^2+n+n^2+2n^2+n^2+3n}{3n^2} \approx \frac{6n^2}{3n^2} = 2$

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

Solución:

```
double fpar(double x[], double y[], double z[], double v[], double w[]) {
   double r1, res;
   #pragma omp parallel
   {
      #pragma omp sections
      {
         #pragma omp section
         A(x,v);
         #pragma omp section
         { B(y,v,w); C(w,v); }
         #pragma omp section
         r1=D(z,v);
      }
      #pragma omp sections
      {
         #pragma omp section
         E(x,v,w);
         #pragma omp section
         res=F(z,r1);
      }
   }
   return res;
}
```

(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.

Solución: Las 3 secciones de la primera construcción `sections` se tendrían que repartir entre 2 hilos. El mejor caso sería que las tareas A y D se hicieran en hilos diferentes, y las tareas B y C en uno cualquiera de los dos hilos. Por ejemplo, un hilo podría hacer las tareas A, B y C (con un coste de $2n^2 + n^2 + n \approxeq 3n^2$ flops) y el otro hilo la tarea D ($2n^2$ flops). Luego un hilo ejecutaría E (n^2 flops) mientras el otro ejecuta F ($3n$ flops). Por tanto:

$$t_2 = \max(3n^2, 2n^2) + \max(n^2, 3n) = 3n^2 + n^2 = 4n^2 \text{ flops}$$

Teniendo en cuenta que el coste secuencial es la suma del coste de todas las tareas, o sea $6n^2$ flops:

$$S_2 = \frac{6n^2}{4n^2} = \frac{6}{4} = 1{,}5$$

$$E_2 = \frac{1{,}5}{2} = 0{,}75$$

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.

Solución:

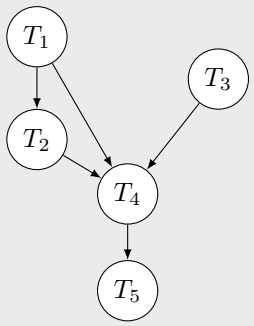
$$t_{\text{tarea4}} = 3$$

$$t_{\text{tarea5}}(n) = \sum_{i=0}^{n-1} 6 = 6n$$

$$t(n) = n + n + 4n + 3 + 6n \approx 12n \text{ flops}$$

(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.

Solución:



El máximo número de tareas que pueden ejecutarse concurrentemente es 2 (tareas 1 y 3 o tareas 2 y 3), así que el grado máximo de concurrencia es 2.

El camino crítico es el de mayor longitud que va de un nodo inicial a un nodo final. En este caso será el formado por las tareas 3, 4 y 5 y su longitud es $L = 4n + 3 + 6n \approx 10n$.

El grado medio de concurrencia se calcula como

$$M = \frac{\sum_{i=1}^{6} t_{\text{tarea i}}}{L} = \frac{12n+3}{10n+3} \approx \frac{12}{10} = 1{,}2$$

(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.

Solución:

```
int ejercicio(double v[n],double x)
{
  int i,j,k=0;
  double a,b,c;
```

```
    #pragma omp parallel
    {
      #pragma omp sections
      {
        #pragma omp section
        {
          a = tarea1(v,x);
          b = tarea2(v,a);
        }
        #pragma omp section
        c = tarea3(v,x);
      }
      #pragma omp single
      x = x + a + b + c;   /* tarea 4 */
      #pragma omp for private(j) reduction(+:k)
      for (i=0; i<n; i++) {   /* tarea 5 */
        j = f(v[i],x);
        if (j>0 && j<4) k++;
      }
    }
    return k;
  }
```

(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.

Solución:

$$t_6(n) = \max\{t_{\text{tarea1}} + t_{\text{tarea2}}, t_{\text{tarea3}}\} + t_{\text{tarea4}} + \sum_{i=0}^{n/6-1} 6 = 4n + 3 + \frac{6n}{6} \approx 5n \text{ flops}$$

$$S_6 = \frac{t(n)}{t_6(n)} = \frac{12n}{5n} = \frac{12}{5} = 2{,}4$$

$$E_6 = \frac{S_6}{p} = \frac{2{,}4}{6} = 0{,}4 \rightarrow 40\,\%$$

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.

Solución: La paralelización del código anterior consiste en paralelizar los bucles internos. El bucle `while` exterior no es paralelizable, mientras que los internos sí lo son si nos fijamos en las dependencias de datos. Se trataría simplemente de añadir la siguiente directiva

```
#pragma omp parallel for private(a)
```

antes de cada bucle `for`. La variable `a` debe ser privada, las demás son compartidas excepto la variable de iteración `i` que es privada por defecto.

La variable `ordenado` es compartida y todos los hilos acceden a ella para escribirla. Esto debería obligar a protegerla en una sección crítica (o una operación de reducción). Sin embargo, en este caso no es necesario.

(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`.

Solución: La manera de contar el número de intercambios consiste en utilizar una variable contador (`int cont=0`). Esta variable ha de ser privada a cada hilo, agregando la cuenta total a la salida del bucle mediante una operación de reducción. La directiva en los bucles sería la siguiente:

```
#pragma omp parallel for private(a) reduction(+:cont)
```

añadiendo en el cuerpo del bucle (dentro del `if`) la instrucción `cont++`.

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 macro `MAX` devuelve el máximo de dos números.

Solución: El uso del vector `ind` hace que varias iteraciones del bucle puedan acabar accediendo a un mismo elemento del vector `x`. Por esta razón, el cálculo del máximo requiere de una sección crítica para evitar condiciones de carrera.

```
void f(int n, double v[], double x[], int ind[])
{
   int i;
   double aux;
   #pragma omp parallel for private(aux)
   for (i=0; i<n; i++) {
      aux=f2(v[i]);
      if (aux>x[ind[i]]) {
         #pragma omp critical
         if (aux>x[ind[i]]) {
            x[ind[i]] = aux;
         }
      }
   }
}
```

La evaluación de la función `f2` se guarda en una variable auxiliar para evitar ser realizada varias veces. Además, sería muy ineficiente realizar la evaluación dentro de la sección crítica, puesto que se haría de forma secuencial.

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.

Solución: En caso de varias ocurrencias, el código proporcionado devuelve la última posición, lo que puede obtenerse en el código paralelo mediante el máximo.

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

   #pragma omp parallel for reduction(max:pos)
   for (i=0; i<n; i++)
      if (x[i]==valor)
         pos=i;
```

```
   return pos;
}
```

La solución anterior no funciona en versiones de OpenMP anteriores a la 3.1. Una solución alternativa es:

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

   #pragma omp parallel for
   for (i=0; i<n; i++)
      if (x[i]==valor)
         if (i>pos)
            #pragma omp critical
            if (i>pos)
               pos=i;

   return pos;
}
```

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.

Solución:

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

double infNorm(double A[DIMN][DIMN], int n)
{
  int i,j;
  double s,norm=0;

  #pragma omp parallel for private(j,s)
  for (i=0; i<n; i++) {
    s = 0;
    for (j=0; j<n; j++)
      s += fabs(A[i][j]);
    if (s>norm)
      #pragma omp critical
      if (s>norm)
        norm = s;
  }
  return norm;
}
```

La paralelización la situamos en el bucle más externo para una mayor granularidad y menor tiempo de sincronización.

La paralelización genera una condición de carrera en la actualización del máximo de las sumas por filas. Para evitar errores en las actualizaciones simultáneas se utiliza una sección crítica que evita la modificación concurrente de `norm`. Para evitar una excesiva secuencialización se incluye la sección crítica dentro de la comprobación del máximo, repitiéndose la comprobación dentro de la sección crítica para asegurar que sólo se modifica el valor de `norm` si es un valor superior al acumulado.

Una solución alternativa sería modificar la directiva `parallel` como se muestra a continuación (haciéndose innecesaria la directiva `critical`). Sin embargo, esta variante no sería válida en versiones antiguas de OpenMP (anteriores a 3.1), pues no se permitía una reducción con el operador `max`.

```
#pragma omp parallel for private(j,s) reduction(max:norm)
```

(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.

Solución: $t(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 2 = 2n^2$ flops $\qquad t(n,p) = \sum_{i=0}^{\frac{n}{p}-1}\sum_{j=0}^{n-1} 2 = 2\frac{n^2}{p}$ flops

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

Solución: $S(n,p) = \frac{t(n)}{t(n,p)} = p \qquad E(n,p) = \frac{S(n,p)}{p} = 1$
La aceleración es la máxima esperable y la eficiencia es del 100 %, lo que indica un aprovechamiento máximo de los 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.

Solución:

```
double prod1(double v[], int n)
{
  double p=1;
  int i;
  #pragma omp parallel for reduction(*:p)
  for (i=0;i<n;i++)
    p *= v[i];
  return p;
}
```

(b) Sin utilizar reducción.

Solución: En este apartado presentamos dos implementaciones distintas, comentando cuál es la más eficiente.

```
double prod2(double v[], int n)
{
  double p=1;
  int i;
  #pragma omp parallel for
  for (i=0;i<n;i++) {
    #pragma omp atomic
    p *= v[i];
  }
  return p;
}

double prod3(double v[], int n)
{
  double p=1;
  int i;
  #pragma omp parallel
  {
    int ppriv=1;   /* ppriv es una variable privada */
    #pragma omp for nowait
    for (i=0;i<n;i++)
      ppriv *= v[i];
    #pragma omp atomic
    p *= ppriv;
  }
```

```
    return p;
  }
```

La implementación `prod3` es más eficiente que la implementación `prod2`, pues el número de operaciones `atomic` realizadas por `prod3` es menor.

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`.

Solución:

```
int cmp(int n, double x[], double y[], int z[])
{
  int i, v, equal=0;
  double aux;
  #pragma omp parallel for private(aux,v) reduction(+:equal)
  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;
}
```

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

Solución: La solución es realizar una región paralela en la que se reparten las iteraciones de forma manual en base al número de hilos y al identificador de hilo. En este caso no se nos permite utilizar la cláusula `reduction`; una posible alternativa sería proteger el acceso a la variable compartida `equal` con una construcción `atomic`.

```
int cmp(int n, double x[], double y[], int z[])
{
  int i, v, equal=0, yo, nh;
```

```
    double aux;
    #pragma omp parallel private(aux,i,v,yo)
    {
      nh = omp_get_num_threads();  /* numero de hilos */
      yo = omp_get_thread_num();   /* identificador de hilo */
      for (i=yo; i<n; i+=nh) {
        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)
          #pragma omp atomic
          equal++;
      }
    }
    return equal;
  }
```

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.

Solución: El uso del vector `ind` hace que varias iteraciones del bucle `i` puedan acabar actualizando un mismo elemento del vector `x`. Por esta razón, esa actualización debe hacerse en exclusión mutua, mediante el uso de la directiva `atomic`.

```
  #pragma omp parallel for private(s,j)
  for (i=0; i<n; i++) {
     s = 0;
     for (j=0; j<i; j++) {
        s += A[i][j]*b[j];
     }
     c[i] = s;
     #pragma omp atomic
     x[ind[i]] += s;
  }
```

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

Solución:

```
for (i=0; i<n; i++) {
   s = 0;
   #pragma omp parallel for reduction(+:s)
   for (j=0; j<i; j++) {
      s += A[i][j]*b[j];
   }
   c[i] = s;
   x[ind[i]] += s;
}
```

(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?

Solución: El bucle `j` hace `i` iteraciones, con lo que las iteraciones del bucle `i` serán más costosas cuanto mayor sea el valor de `i`. Por tanto, una planificación estática con un único *chunk* por hilo no sería adecuada. Sería mejor una planificación cíclica (estática con tamaño de `chunk`=1), dinámica o *guided*.

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.

Solución: El fragmento de código que calcula el máximo es independiente del fragmento que calcula el mínimo, por lo que se podría pensar en una paralelización basada en `sections`, pero se descarta esta opción porque solo se aprovecharían 2 hilos, y el enunciado pide una paralelización eficiente para cualquier número de hilos. Por tanto, se hace una paralelización de cada uno de los bucles. Dado que el primer bucle es independiente del segundo, hacemos uso de la cláusula `nowait` para evitar una sincronización entre ambos.

La solución utilizando reducciones sería la siguiente.

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

  mx = a[0];
  mn = a[0];

  #pragma omp parallel
  {
    #pragma omp for reduction(max:mx) nowait
    for (i=1;i<n;i++) {
      if (mx<a[i]) mx=a[i];
    }
    #pragma omp for reduction(min:mn)
    for (i=1;i<n;i++) {
      if (mn>a[i]) mn=a[i];
    }
    factor = mx-mn;
    #pragma omp for
    for (i=0;i<n;i++) {
      a[i]=(a[i]-mn)/factor;
    }
  }
}
```

Como alternativa a las reducciones, se pueden usar secciones críticas, aunque sería menos eficiente.

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

  mx = a[0];
  mn = a[0];

  #pragma omp parallel
  {
    #pragma omp for nowait
    for (i=1;i<n;i++) {
      if (mx<a[i])
        #pragma omp critical (mx)
        if (mx<a[i]) mx=a[i];
    }
    #pragma omp for
    for (i=1;i<n;i++) {
      if (mn>a[i])
        #pragma omp critical (mn)
        if (mn>a[i]) mn=a[i];
    }
    factor = mx-mn;
    #pragma omp for
```

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

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

Solución:

```
      ...
      #pragma omp parallel
      {
        #pragma omp single
        printf("Num procs: %d\n", omp_get_num_threads());

        #pragma omp for nowait
        ...
```

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.

Solución:

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

```
    suma=0;
    #pragma omp parallel
    {
      #pragma omp for reduction(+:suma)
      for (i=0;i<n;i++)
        suma = suma + v[i]*v[i];
      norma = sqrt(suma);

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

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

En el tercero de los bucles paralelizados se calcula un máximo y su posición. Las variables compartidas `pos_max` y `max` pueden ser actualizadas simultáneamente en diferentes iteraciones, por lo que se requiere exclusión mutua en el acceso a ellas (construcción `critical`). Además, la repetición de la instrucción

```
if (aux > max)
```

antes de la sección crítica tiene por objeto mejorar la eficiencia, evitando entrar en esta sección en casos en que no es necesario.

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

Solución: No. En el primero no, porque hasta que no acaben todos los hilos no se puede calcular la norma correctamente. En el segundo no, porque el tercer bucle necesita los valores de `v[i]` actualizados del segundo. En el tercero tampoco, porque no hay nada paralelo detrás.

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

Solución: Se añadiría en las directivas `for` una cláusula de planificación apropiada, como puede ser `schedule(static,2)` o `schedule(dynamic,2)` que indique que se repartan las iteraciones entre los hilos de 2 en 2.

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.

Solución: La forma más sencilla de implementarlo es mediante una reducción (suponiendo OpenMP versión 3.1 o posterior). Además, hay que sincronizar el acceso concurrente a la variable `saldos`.

```
double transferencias(double saldos[], int origenes[],
       int destinos[], double cantidades[], int n)
{
   int i, i1, i2;
   double dinero, maxtransf=0;
   #pragma omp parallel for private(i1,i2,dinero) \
               reduction(max:maxtransf)
   for (i=0; i<n; i++) {
      i1 = origenes[i];
      i2 = destinos[i];
      dinero = cantidades[i];
      #pragma omp atomic
      saldos[i1] -= dinero;
      #pragma omp atomic
      saldos[i2] += dinero;
      if (dinero>maxtransf) maxtransf = dinero;
   }
   return maxtransf;
}
```

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

Solución: En este caso no sirve la directiva `reduction` y es necesario crear una sección crítica.

```
double transferencias(double saldos[], int origenes[],
       int destinos[], double cantidades[], int n)
```

```
    {
       int i, i1, i2;
       double dinero, maxtransf=0, imax;
       #pragma omp parallel for private(i1,i2,dinero)
       for (i=0; i<n; i++) {
          i1 = origenes[i];
          i2 = destinos[i];
          dinero = cantidades[i];
          #pragma omp atomic
          saldos[i1] -= dinero;
          #pragma omp atomic
          saldos[i2] += dinero;
          if (dinero>maxtransf)
            #pragma omp critical
            if (dinero>maxtransf) {
              maxtransf = dinero;
              imax = i;
            }
       }
       printf("Transferencia con más dinero=%d\n",imax);
       return maxtransf;
    }
```

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.

Solución: El programa se puede dividir en tres tareas:

- Tarea 1: Modificación de la matriz `A` a partir de la propia matriz `A` (primer bucle anidado).
- Tarea 2: Modificación de la matriz `B` a partir de la matriz `A` (segundo bucle anidado).
- Tarea 3: Obtención de `maxi` $= \max_{\substack{0\leq i<N \\ 0\leq j<N}} \{b_{ij}^2\}$, siendo b_{ij} el elemento (i,j) de la matriz `B` (tercer bucle anidado).

Como puede observarse, la Tarea 2 depende de la Tarea 1 y la Tarea 3 de la Tarea 2, por lo que ha de finalizar una tarea antes de iniciarse la siguiente. Para paralelizar el código, paralelizaremos las tres tareas a nivel de bucles:

- Tarea 1: Como hay una dependencia en las iteraciones del bucle cuya variable iteradora es `i` (los valores de la nueva fila `i` de la matriz `A` dependen de los anteriores valores de la fila `i-1` de la matriz `A`) y siempre conviene paralelizar el más externo, intercambiamos el orden de los bucles paralelizando el más externo (paralelización por columnas).
- Tarea 2: Paralelizamos directamente el bucle más externo (obtención en paralelo de las filas de la matriz `B`).
- Tarea 3: Paralelizamos directamente el bucle más externo. Para obtener el valor `maxi` se puede realizar una reducción sobre `maxi` o usar regiones críticas.

A continuación se muestra la primera posibilidad:

```
double funcionp(double A[N][N],double B[N][N]) {
  int i,j;
  double aux, maxi;
  #pragma omp parallel for private(i)
  for (j=0; j<N; j++)
    for (i=1; i<N; i++)
      A[i][j] = 2.0+A[i-1][j];
  #pragma omp parallel for private(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;
  #pragma omp parallel for private(j,aux) reduction(max:maxi)
  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;
}
```

La otra posibilidad sería cambiar el tercer bucle anidado por el siguiente:

```
  #pragma omp parallel for private(j,aux)
  for (i=0; i<N; i++)
    for (j=0; j<N; j++) {
      aux = B[i][j]*B[i][j];
```

```
        if (aux>maxi)
          #pragma omp critical
          {
            if (aux>maxi) maxi = aux;
          }
      }
```

(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.

Solución:

$$t(N) = \sum_{i=1}^{N-1}\sum_{j=0}^{N-1} 1 + \sum_{i=0}^{N-2}\sum_{j=0}^{N-2} 1 + \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} 1 \approx N^2 + N^2 + N^2 = 3N^2 \text{ flops.}$$

$$t(N,p) = \sum_{j=0}^{N/p-1}\sum_{i=1}^{N-1} 1 + \sum_{i=0}^{N/p-2}\sum_{j=0}^{N-2} 1 + \sum_{i=0}^{N/p-1}\sum_{j=0}^{N-1} 1 \approx \frac{N^2}{p} + \frac{N^2}{p} + \frac{N^2}{p} = \frac{3N^2}{p} \text{ flops.}$$

$$S(N,p) = \frac{t(N)}{t(N,p)} = \frac{3N^2}{\frac{3N^2}{p}} = p.$$

$$E(N,p) = \frac{S(N,p)}{p} = \frac{p}{p} = 1.$$

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.

Solución: En la paralelización del segundo fragmento de código, hay que proteger el acceso a las variables `k` y `posiciones` mediante una sección crítica, para evitar condiciones de carrera.

```
int funcion(double A[N][N],double posiciones[][2])
{
  int i,j,k=0;
  double maximo;
  /* Calculamos el máximo */
  maximo = A[0][0];
  #pragma omp parallel
  {
    #pragma omp for private(j) reduction(max:maximo)
    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 */
    #pragma omp for private(j)
    for (i=0;i<N;i++) {
      for (j=0;j<N;j++) {
        if (A[i][j] == maximo) {
          #pragma omp critical
          {
            posiciones[k][0] = i;
            posiciones[k][1] = j;
            k = k+1;
          }
        }
      }
    }
  }
  return k;
}
```

(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`.

Solución: Incorporaríamos las variables `id` y `num_maximos`.

```
int funcion(double A[][N],double posiciones[][2])
{
  int i,j,k=0,id,num_maximos;
  double maximo;
  /* Calculamos el máximo */
  maximo = A[0][0];
  #pragma omp parallel private(id,num_maximos)
  {
    #pragma omp for private(j) reduction(max:maximo)
    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 */
      id = omp_get_thread_num();
      num_maximos = 0;
      #pragma omp for private(j)
      for (i=0;i<N;i++) {
        for (j=0;j<N;j++) {
          if (A[i][j] == maximo) {
            #pragma omp critical
            {
              posiciones[k][0] = i;
              posiciones[k][1] = j;
              k = k+1;
            }
            num_maximos++;
          }
        }
      }
      printf("Hilo %d encuentra %d máximos\n",id,num_maximos);
    }
    return k;
  }
```

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];
  }
  ...
}
```

Solución: En el primer bucle, tenemos que dos o más iteraciones pueden corresponder a actuaciones de un mismo jugador, en cuyo caso actualizarían el mismo elemento de los vectores `suma_valoracion` y `num_partidos`. Para evitar problemas de condiciones de carrera, la actualización de esos vectores deberá hacerse por tanto en exclusión mutua, mediante la cláusula `atomic`.

```
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];
  ...
  #pragma omp parallel
  {
    #pragma omp for private(jugador,puntos,rebotes,tapones)
    for (i=0;i<NA;i++) {
      jugador = M[i][0];
      puntos  = M[i][1];
      rebotes = M[i][2];
      tapones = M[i][3];
      #pragma omp atomic
      suma_valoracion[jugador] += puntos+1.5*rebotes+2*tapones;
      #pragma omp atomic
      num_partidos[jugador]++;
      if (puntos>max_pts) {
        #pragma omp critical
        if (puntos>max_pts) {
          max_pts  = puntos;
          max_anot = jugador;
        }
      }
    }
    #pragma omp master
    printf("Maximo anotador %d (%d puntos)\n",max_anot,max_pts);
    #pragma omp for
    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);
}
```

Solución:

```
/* 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;
  #pragma omp parallel
  {
    #pragma omp for private(i,p)
    for (k = 0; k < nv; k++) {
      i = indice[k]; p = puntos[k];
      #pragma omp atomic
```

```
        totales[i] += p;
        if (p > max2)
          #pragma omp critical
          if (p > max2)
            if (p > max1) { max2 = max1; max1 = p; } else max2 = p;
      }
      #pragma omp single nowait
      printf("Las dos puntuaciones más altas han sido %d y %d.\n",max1,max2);
      #pragma omp for private(t) reduction(+:pasan,total)
      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.

Solución:

```
void facturaciones(int n, int usuarios[], int canciones[], float precios[],
                   float facturacion[], int ndescargas[])
{
```

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

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

Solución: No sería válido emplear la directiva `nowait`, puesto que el segundo bucle sólo puede comenzar cuando ya ha terminado el primero y hemos completado el número de veces que se ha descargado cada canción.

(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.

Solución:

```
void facturaciones(int n, int usuarios[], int canciones[], float precios[],
                   float facturacion[], int ndescargas[])
{
  int i,u,c,mejor_cancion=0;
  float p;
  int myid,descargas_hilo;
  #pragma omp parallel private(myid,descargas_hilo)
  {
    myid = omp_get_thread_num();
    descargas_hilo = 0;
    #pragma omp for private(u,c,p)
    for (i=0;i<n;i++) {
       u = usuarios[i];
       c = canciones[i];
       p = precios[c];
       #pragma omp atomic
       facturacion[u] += p;
```

```
            #pragma omp atomic
            ndescargas[c]++;
            descargas_hilo++;
        }
        printf("El hilo %d ha gestionado %d descargas\n",myid, descargas_hilo);
        #pragma omp for
        for (i=0;i<NC;i++) {
            if (ndescargas[i]>ndescargas[mejor_cancion])
              #pragma omp critical
              if (ndescargas[i]>ndescargas[mejor_cancion])
                mejor_cancion = i;
        }
      }
      printf("La canción %d es la más descargada\n",mejor_cancion);
    }
```