1 Paral·lelisme de bucles

Qüestió 1–1

Segons les condicions de Bernstein, indica els tipus de dependències de dades existents entre les diferents iteracions en els casos que es presenten a continuació. Justifica si es poden eliminar o no aquestes dependències de dades, eliminant-les en cas que siga possible.

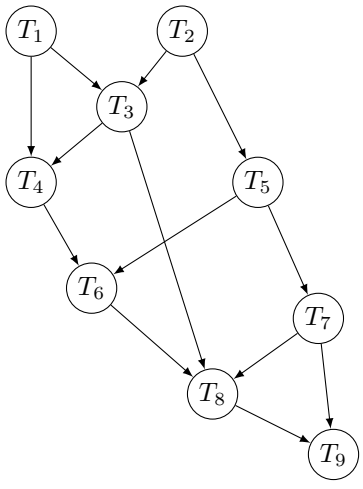
(a)
```
for (i=1;i<N-1;i++) {
  x[i+1] = x[i] + x[i-1];
}
```

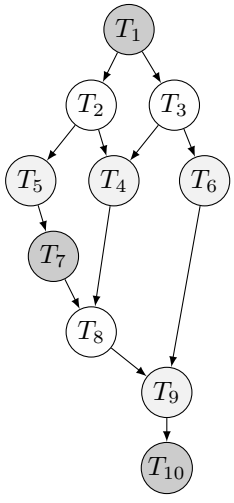
(b)
```
for (i=0;i<N;i++) {
  a[i] = a[i] + y[i];
  x = a[i];
}
```

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

Qüestió 1–2

Donats els següents grafs de dependències de tasques:





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) Per al graf de l'esquerra, indica quina seqüència de nodes del graf constitueix el camí crític. Calcula la longitud del camí crític i el grau mitjà de concurrència. Nota: no s'ofereix informació de costs, es pot suposar que totes les tasques tenen el mateix cost.

(b) Repeteix l'apartat anterior per al graf de la dreta. Nota: en aquest cas el cost de cada tasca ve donat en flops (per a una grandària de problema n) segons la taula mostrada.

Qüestió 1–3

El següent codi seqüencial implementa el producte d'una matriu B de dimensió $N \times N$ per un vector c de dimensió 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) Realitza una implementació paral·lela mitjançant OpenMP del codi anterior.

(b) Calcula els costos computacionals en flops de les implementacions seqüencial i paral·lela, suposant que el nombre de fils p és un divisor de N.

(c) Calcula l'speedup i l'eficiència del codi paral·lel.

Qüestió 1–4

Donada la següent funció:

```
double funcio(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 el seu cost teòric (en flops).

(b) Paral·lelitza-la usant OpenMP. Per què ho fas així? Es valoraran més aquelles solucions que siguen més eficients.

(c) Indica el speedup que podrà obtenir-se amb p processadors suposant M i N múltiples exactes de p.

(d) Indica una cota superior del speedup (quan p tendeix a infinit) si no es paral·lelitzara la part que calcula la suma (és a dir, només es paral·lelitza la primera part i la segona s'executa seqüencialment).

Qüestió 1–5

Donada la següent funció:

```
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 com es paral·lelitzaria mitjançant OpenMP cadascun dels tres bucles. Quina de les tres formes de paral·lelitzar serà la més eficient i per què?

(b) Suposant que es paral·lelitza el bucle més extern, indica els costos a priori seqüencial i paral·lel, en flops, i el speedup suposant que el nombre de fils (i processadors) coincideix amb n.

(c) Afig les línies de codi necessàries perquè es mostre en pantalla el nombre d'iteracions que ha realitzat el fil 0, suposant que es paral·lelitza el bucle més extern.

Qüestió 1–6

Implementa un programa paral·lel utilitzant OpenMP que complisca els següents requisits:

- Demane per teclat un nombre enter positiu n.
- Calcule en paral·lel la suma dels primers n nombres naturals, utilitzant per a açò una distribució que repartisca els nombres a sumar de 2 en 2, sent 6 el nombre de fils usat.
- Al final del programa haurà d'imprimir en pantalla l'identificador del fil que ha sumat l'últim nombre (n) i la suma total calculada.

Qüestió 1–7

Volem paral·lelitzar de forma eficient la següent funció mitjantçant 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) Paral·lelitza-la de forma eficient.

(b) Calcula el cost computacional d'una iteració del bucle k. Calcula el cost computacional de la versió paral·lela (assumint que es divideix el nombre d'iteracions de forma exacta entre el nombre de fils) i l'speed-up.

Qüestió 1–8

Donada la següent funció:

```
#define N 6000
#define PASSOS 6

double funcio1(double A[N][N], double b[N], double x[N])
{
  int i, j, k, n=N, passos=PASSOS;
  double max=-1.0e308, q, s, x2[N];
  for (k=0;k<passos;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) Paral·lelitza el codi mitjançant OpenMP. Per què ho fas així? Es valoraran més aquelles solucions que siguen més eficients.

(b) Indica el cost teòric (en flops) que tindria una iteració del bucle `k` del codi seqüencial.

(c) Considerant una única iteració del bucle `k` (`PASSOS`=1), indica l'speedup i l'eficiència que podrà obtenir-se amb p fils, suposant que hi ha tants nuclis/processadors com fils i que `N` és un múltiple exacte de p.

Qüestió 1–9

Donada la següent funció:

```
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) Fes una versió paral·lela basada en la paral·lelització del bucle: `i`.

(b) Fes una versió paral·lela basada en la paral·lelització del bucle `j`.

(c) Calcula el temps d'execució seqüencial a priori d'una sola iteració del bucle `i`, així com el temps d'execució seqüencial de la funció completa. Suposa que el cost d'una comparació de nombres en coma flotant és 1 flop.

(d) Indica si hi hauria un bon equilibri de càrrega si s'utilitza la clàusula `schedule(static)` en la paral·lelització del primer apartat. Raona la resposta.

Qüestió 1–10

Donada la següent funció:

```
double quad_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) Paral·lelitza el codi anterior de forma eficient mitjançant OpenMP. De les possibles planificacions, quines podrien ser les més eficients? Justifica la resposta.

(b) Calcula el cost de l'algoritme seqüencial en flops.

Qüestió 1–11

Donada la següent funció:

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

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

(a) Paral·lelitzeu (eficientment) el bucle `i` mitjançant OpenMP.

(b) Paral·lelitzeu (eficientment) els dos bucles `j` mitjançant OpenMP.

(c) Calculeu el cost seqüencial del codi original.

(d) Suposant que paral·lelitzem només el primer bucle `j`, calculeu el cost paral·lel corresponent. Obteniu també el speedup i l'eficiència en el cas de que es dispose de `N` processadors.

2 Regions paral·leles

Qüestió 2–1

Donada la següent funció, que cerca un valor en un vector, paral·lelitza-la usant OpenMP. Igual que la funció de partida, la funció paral·lela haurà d'acabar la cerca tan aviat com es trobe l'element cercat.

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

Qüestió 2–2

Donat un vector v de n elements, la següent funció calcula la seua 2-norma $\|v\|$, definida com:

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

```
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) Paral·lelitzar la funció anterior mitjançant OpenMP, seguint el següent esquema:

- En una primera fase, es vol que cada fil calcule la suma de quadrats d'un bloc de n/p elements del vector v (on p és el nombre de fils). Cada fil deixarà el resultat en la posició corresponent d'un vector `sumes` de p elements. Es pot assumir que el vector `sumes` ja ha sigut creat (encara que no inicialitzat).
- En una segona fase, un dels fils calcularà la norma del vector, a partir de les sumes parcials emmagatzemades en el vector `sumes`.

(b) Paral·lelitzar la funció de partida mitjançant OpenMP, usant una altra aproximació diferent de la de l'apartat anterior.

(c) Calcular el cost a priori de l'algorisme seqüencial de partida. Raonar quin seria el cost de l'algorisme paral·lel de l'apartat a, i el speedup obtingut.

Qüestió 2–3

Donada la següent funció:

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

Paral·lelitza-la, fent a més que cada fil escriga un missatge indicant el seu número de fil i quantes iteracions ha processat. Es vol mostrar un sol missatge per cada fil.

Qüestió 2–4

Donada la següent funció:

```
void normalitza(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) Paral·lelitza-la amb OpenMP usant dues regions paral·leles.

(b) Paral·lelitza-la amb OpenMP usant una única regió paral·lela que englobe a tots els bucles. En aquest cas, tindria sentit utilitzar la clàusula `nowait`? Justifica la resposta.

Qüestió 2–5

Donada la següent funció:

```
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) Paral·lelitza-la eficientment mitjançant OpenMP, utilitzant una sola regió paral·lela.

(b) Calcula el nombre de flops de la funció inicial i de la funció paral·lelitzada.

(c) Determina l'speedup i l'eficiència.

Qüestió 2–6

Paral·lelitza el següent fragment de codi mitjançant seccions d'OpenMP. El segon argument de les funcions `fun1`, `fun2` i `fun3` és d'entrada-sortida, és a dir, aquestes funcions utilitzen i modifiquen 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;
```

Qüestió 2–7

Donada la següent funció:

```
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 argument de totes les funcions usades és d'eixida i la resta d'arguments són arguments d'entrada. Per exemple, `f1(a,b)` és una funció que a partir del vector `b` modifica el vector `a`.

(a) Dibuixa el graf de dependències de tasques i indica almenys 2 tipus diferents de dependències que apareguen en aquest problema.

(b) Paral·lelitza la funció per mitjà de directives OpenMP.

(c) Suposant que totes les funcions tenen el mateix cost i que es disposa d'un nombre de processadors arbitrari, quin serà l'speedup màxim possible? Es podria millorar l'speedup utilitzant replicació de dades?

Qüestió 2–8

En la següent funció, `T1`, `T2`, `T3` modifiquen `x`, `y`, `z`, respectivament.

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

   T1(x,n);    /* Tasca T1 */
   T2(y,n);    /* Tasca T2 */
   T3(z,n);    /* Tasca T3 */

   /* Tasca 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;
   }

   /* Tasca 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;
   }

   /* Tasca T6 */
   a=s1/s2;
   res=0;
   for (i=0; i<n; i++) res+=a*z[i];
   return res;
}
```

(a) Dibuixa el graf de dependències de les tasques.

(b) Realitza una paral·lelització per mitjà d'OpenMP a nivell de tasques (no de bucles), en base al graf de dependències.

(c) Indica el cost a priori de l'algoritme seqüencial, el de l'algoritme paral·lel i l'speedup resultant. Suposa que el cost de les tasques 1, 2 i 3 és de $2n^2$ flops cadascuna.

Qüestió 2–9

Donat el següent fragment de codi:

```
minx = minim(x,n);          /* T1 */
maxx = maxim(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) Dibuixa el graf de dependències de les tasques, tenint en compte que les funcions `minim` i `maxim` no modifiquen els seus arguments, mentre que les altres funcions modifiquen només el seu primer argument.

(b) Paral·lelitza el codi mitjançant OpenMP.

(c) Si el cost de les tasques és de n flops, excepte el de la tasca 4 que és de $2n$ flops, indica la longitud del camí crític i el grau mitjà de concurrència. Obteniu l'speedup i l'eficiència de la implementació de l'apartat anterior, si s'executara amb 5 processadors.

Qüestió 2–10

Es vol paral·lelitzar el següent programa mitjançant OpenMP, on `genera` és una funció prèviament definida en un altre lloc.

```
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;
}
```

```
/* fragment 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) Paral·lelitza el codi de forma eficient a nivell de bucles.

(b) Dibuixa el graf de dependències de tasques, segons la numeració de tasques indicada en el codi.

(c) Paral·lelitza el codi de forma eficient a nivell de tasques, a partir del graf de dependències anterior.

(d) Trau el temps seqüencial (assumeix que una crida a les funcions `genera` i `fabs` costa 1 flop) i el temps paral·lel per a cadascuna de les dues versions assumint que hi ha 3 processadors. Calcular l'speed-up en cada cas.

Qüestió 2–11

Paral·lelitza mitjançant OpenMP el següent fragment de codi, on `f` i `g` són dues funcions que prenen

3 arguments de tipus `double` i retornen un `double`, i `fabs` és la funció estàndard que retorna el valor absolut d'un `double`.

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

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);
```

Qüestió 2–12

Tenint en compte la definició de les següents funcions:

```
/* producte 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;
    }
  }
}
```

```
/* simetritza una matriu com A+A' */
void simetritza(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;
    }
  }
}
```

es pretén paral·lelitzar el següent codi:

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

(a) Realitza una paral·lelització basada en els bucles.

(b) Dibuixa el graf de dependències de tasques, considerant en aquest cas que les tasques són cadascuna de les crides a `matmult` i `simetritza`. Indica quin és el grau màxim de concurrència, la longitud

del camí crític i el grau mitjà de concurrència. Nota: per a determinar aquests últims valors, és necessari obtenir el cost en flops d'ambdues funcions.

(c) Realitza la paral·lelització basada en seccions, a partir del graf de dependències anterior.

Qüestió 2–13

Donada la següent funció:

```
void updatemat(double A[N][N])
{
  int i,j;
  double s[N];
  for (i=0; i<N; i++) {     /* suma de files */
    s[i] = 0.0;
    for (j=0; j<N; j++)
      s[i] += A[i][j];
  }
  for (i=1; i<N; i++)       /* suma prefixa */
    s[i] += s[i-1];
  for (j=0; j<N; j++) {     /* escalat de columnes */
    for (i=0; i<N; i++)
      A[i][j] *= s[j];
  }
}
```

(a) Indica el cost teòric (en flops) de la funció proporcionada.

(b) Paral·lelitza-la amb OpenMP amb una única regió paral·lela.

(c) Indica l'speedup que es podrà obtindre amb p processadors suposant que N és múltiple exacte de p.

Qüestió 2–14

Donada la següent funció:

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

  omplir(A,B);              /* T1 */
  a = calculs(A);           /* T2 */
  b = calculs(B);           /* T3 */
  x = suma_menors(B,a);     /* T4 */
  y = suma_en_rang(B,a,b); /* T5 */
  z = x + y;                /* T6 */
  return z;
}
```

La funció `omplir` rep dos matrius i les ompli amb valors generats internament. Els paràmetres de la resta de funcions són només d'entrada (no es modifiquen). Les funcions `omplir` i `suma_en_rang` tenen un cost de $2n^2$ flops cadascuna ($n =$ `N`), mentres que el cost de cadascuna de les altres funcions és n^2 flops.

(a) Dibuixa el graf de dependències i indica el seu grau màxim de concurrència, un camí crític i la seua longitud, i el grau mitjà de concurrència.

(b) Paral·lelitza la funció amb OpenMP.

(c) Calcula el temps d'execució seqüencial, el temps d'execució paral·lel, l'speed-up i l'eficiència del codi de l'apartat anterior, suposant que es treballa amb 3 fils.

Qüestió 2–15

Es vol paral·lelitzar el següent codi de processament d'imatges, que rep com a entrada 4 imatges similars (per exemple, fotogrames d'un vídeo `f1`, `f2`, `f3`, `f4`) i torna dos imatges resultat (`r1`, `r2`). Els píxels de la imatge es representen com a nombres en coma flotant (`image` es un nou tipus de dades consistent en una matriu de `N`×`M` `double`s).

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

void processa(image f1,image f2,image f3,image f4,image r1,image r2)
{
  image d1,d2,d3;
  difer(f2,f1,d1);          /* Tasca 1 */
  difer(f3,f2,d2);          /* Tasca 2 */
  difer(f4,f3,d3);          /* Tasca 3 */
  suma(d1,d2,d3,r1);        /* Tasca 4 */
  difer(f4,f1,r2);          /* Tasca 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) Dibuixa el graf de dependències de tasques, i indica quin seria el grau màxim i mitjà de concurrència, tenint en compte el cost en flops (suposa que `fabs` no realitza cap flop).

(b) Paral·lelitza la funció `processa` mitjançant OpenMP, sense modificar `difer` i `suma`.

Qüestió 2–16

En la següent funció, cap de les funcions cridades (`A`,`B`,`C`,`D`) modifica els seus paràmetres:

```
double calculs_matricials(double mat[n][n])
{
  double x,y,z,aux,total;
  x = A(mat);          /* tasca A, cost: 3 n^2         */
  aux = B(mat);        /* tasca B, cost: n^2           */
  y = C(mat,aux);      /* tasca C, cost: n^2           */
  z = D(mat);          /* tasca D, cost: 2 n^2         */
  total = x + y + z;   /* tasca E (calcula tu mateix el seu cost) */
  return total;
}
```

(a) Dibuixa el seu graf de dependències i indica el grau màxim de concurrència, la longitud del camí crític, mostrant un camí crític, i el grau mitjà de concurrència.

(b) Paral·lelitza-la amb OpenMP.

(c) Calcula el temps seqüencial en flops. Suposant que s'executa amb 2 fils, calcula el temps paral·lel, l'speedup i l'eficiència, en el millor dels casos.

(d) Modifica el codi paral·lel perquè es mostre per pantalla (una sola vegada) el nombre de fils amb què s'ha executat i el temps d'execució utilitzat en segons.

Qüestió 2–17

Donada la següent funció:

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

(a) Fes una versió paral·lela basada en la paral·lelització del bucle `i` amb OpenMP.

(b) Fes una altra versió paral·lela basada en la paral·lelització dels bucles `j` i `j2` (de forma eficient per a qualsevol nombre de fils).

(c) Calcula el cost (temps d'execució) del codi seqüencial.

(d) Per a cadascun dels tres bucles, justifica si cabria esperar diferències de prestacions depenent de la planificació emprada al paral·lelitzar el bucle. Si és així, indica quines planificacions serien millor per al bucle corresponent.

Qüestió 2–18

Es desitja paral·lelitzar la següent funció, on `llig_dades` modifica els seus tres arguments i `f5` llig i escriu els seus dos primers arguments. La resta de funcions no modifiquen els seus arguments.

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

(a) Dibuixa el graf de dependències de les diferents tasques que componen la funció.

(b) Paral·lelitza la funció eficientment amb OpenMP.

(c) Calcula l'speedup i l'eficiència si emprem 3 processadors.

(d) A partir dels costos de cada tasca reflectits en el codi de la funció, obtín la longitud del camí crític i el grau mitjà de concurrència.

Qüestió 2–19

Donada la següent funció, on saben que totes les funcions a les què es crida modifiquen només el vector que reben com a primer argument:

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

(a) Dibuixa el graf de dependències. Identifica un camí crític i indica la seua longitud. Calcula el grau mitjà de concurrència.

(b) Implementa una versió paral·lela eficient de la funció.

(c) Suposant que el codi de l'apartat anterior s'executa amb 2 fils, calcula el temps d'execució paral·lel, l'speed-up i l'eficiència, en el millor dels casos. Raona la resposta.

Qüestió 2–20

En la següent funció, cap de les funcions a les que es crida modifiquen els seus paràmetres.

```
int exercici(double v[n],double x)
{
  int i,j,k=0;
  double a,b,c;
  a = tasca1(v,x);   /* tasca 1, cost n flops */
  b = tasca2(v,a);   /* tasca 2, cost n flops */
  c = tasca3(v,x);   /* tasca 3, cost 4n flops */
  x = x + a + b + c;   /* tasca 4 */
  for (i=0; i<n; i++) {   /* tasca 5 */
    j = f(v[i],x);   /* cada crida a esta funció costa 6 flops */
    if (j>0 && j<4) k++;
  }
  return k;
}
```

(a) Calcula el temps d'execució seqüencial.

(b) Dibuixa el graf de dependències a nivell de tasques (considerant la tasca 5 com indivisible) i indica el grau màxim de concurrència, la longitud del camí crític i el grau mitjà de concurrència.

(c) Paral·lelitza-la de forma eficient utilitzant una sola regió paral·lela. A més de realitzar en paral·lel aquelles tasques que es puguen, paral·lelitza també el bucle de la tasca 5.

(d) Suposant que s'executa amb 6 fils (i que n és un múltiple exacte de 6), calcula el temps d'execució paral·lel, l'speed-up i l'eficiència.

Qüestió 2–21

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

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

Donada la definició de les funcions anteriors, es pretén paral·lelitzar el següent codi:

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

(a) Dibuixeu el graf de dependències de tasques. Indiqueu quina és la longitud del camí crític i el grau mitjà de concurrència. Nota: per a determinar estos últims valors, és necessari obtindre el cost en flops d'ambdues funcions. Assumir que `sqrt` costa 5 flops.

(b) Realitza la paral·lelització basada en seccions, a partir de graf de dependències anterior.

Qüestió 2–22

Donada la següent funció:

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

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

(a) Paral·lelitzeu la funció de forma eficient mitjançant OpenMP.

(b) Indiqueu el seu cost paral·lel teòric (en *flops*), assumint que `N` és múltiple del nombre de fils. Per a avaluar el cost considereu el cas pitjor, és a dir, que totes les comparacions siguen certes. A més, suposeu que el cost de comparar dos nombres reals és 1 *flop*.

(c) Modifiqueu el codi perquè cada fil mostre un únic missatge amb el seu índex de fil i el nombre d'elements que ha sumat.

Qüestió 2–23

Siga el següent codi:

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

(a) Obteniu el graf de dependències i expliqueu quin tipus de dependències ocorren entre T_2 i T_3 i entre T_4 i T_7, en cas de que hi haguen.

(b) Calculeu la longitud del camí crític i indiqueu les tasques que el formen.

(c) Implementeu una versió paral·lela el més eficient possible del codi anterior mitjançant seccions, emprant una única regió paral·lela.

(d) Calculeu el speedup i l'eficiència si emprem 4 fils per a executar el codi paral·lelitzat en l'apartat anterior.

3 Sincronització

Qüestió 3–1

Siga el següent codi que permet ordenar un vector `v` de `n` nombres reals ascendentment:

```
int ordenat = 0;
double a;
while( !ordenat ) {
  ordenat = 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;
      ordenat = 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;
      ordenat = 0;
    }
```

```
    }
  }
```

(a) Introduir les directives OpenMP que permeten executar aquest codi en paral·lel.

(b) Modificar el codi per a comptabilitzar el nombre d'intercanvis que es produeixen, és a dir, el nombre de vegades que s'entra en qualsevol de les dues estructures `if`.

Qüestió 3–2

Donada la funció:

```
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]));
   }
}
```

Paral·lelitzar la funció, tenint en compte que `f2` és una funció molt costosa. Es valorarà que la solució aportada siga eficient.

Nota. Assumim que `f2` no té efectes laterals i el seu resultat només depén del seu argument d'entrada. El tipus de retorn de la funció `f2` és `double`. La funció `max` torna el màxim de dos valors.

Qüestió 3–3

Donada la següent funció, la qual 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;
}
```

Es demana paral·lelitzar-la amb OpenMP. En cas de vàries ocurrències del valor en el vector, l'algoritme paral·lel deu tornar el mateix que el seqüencial.

Qüestió 3–4

La infinit-norma d'una matriu $A \in \mathbb{R}^{n \times n}$ es defineix com el màxim de les sumes dels valors absoluts dels elements de cada fila:

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

El següent codi seqüencial implementa aquesta operació per al cas d'una matriu quadrada.

```
#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) Realitza una implementació paral·lela mitjançant OpenMP d'aquest algorisme. Justifica la raó per la qual introdueixes cada canvi.

(b) Calcula el cost computacional (en flops) de la versió original seqüencial i de la versió paral·lela desenvolupada.
Nota: Es pot assumir que la dimensió de la matriu n és un múltiple exacte del nombre de fils p. Es pot assumir que el cost de la funció `fabs` és d'1 flop.

(c) Calcula l'speedup i l'eficiència del codi paral·lel executat en p processadors.

Qüestió 3–5

Donada la següent funció, que calcula el producte dels elements 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 dues funcions paral·leles:

(a) Utilitzant reducció.

(b) Sense utilitzar reducció.

Qüestió 3–6

Es vol paral·lelitzar de forma eficient la següent funció mitjançant 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) Paral·lelitza-la utilitzant construccions de tipus `parallel for`.

(b) Paral·lelitza-la sense usar cap de las següents primitives: `for`, `section`, `reduction`.

Qüestió 3–7

Donat el següent fragment de codi, on el vector d'índexs `ind` conté valors enters entre 0 i $m-1$ (sent m la dimensió de `x`), possiblement amb repeticions:

```
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) Realitza una implementació paral·lela mitjançant OpenMP, en la qual es reparteixen les iteracions del bucle extern.

(b) Realitza una implementació paral·lela mitjançant OpenMP, en la qual es reparteixen les iteracions del bucle intern.

(c) Per a la implementació de l'apartat (a), indica si cal esperar que hi haja diferències de prestacions depenent de la planificació emprada. Si és així, quines planificacions serien millors i per què?

Qüestió 3–8

La següent funció normalitza els valors d'un vector de nombres reals positius de manera que els valors finals queden entre 0 i 1, utilitzant el màxim i el mínim.

```
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) Paral·lelitza el programa amb OpenMP de la manera més eficient possible, mitjançant una única regió paral·lela. Suposeu que el valor de `n` és molt gran i que es vol que la paral·lelització funcione eficientement per a un nombre arbitrari de fils.

(b) Inclou el codi necessari perquè s'imprimisca una sola vegada el nombre de fils utilitzats.

Qüestió 3–9

Donada la següent funció:

```
int funcio(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) Paral·lelitza-la amb OpenMP, usant una única regió paral·lela.

(b) Tindria sentit posar una clàusula `nowait` a algun dels bucles? Per què? Justifica cadascun dels bucles separadament.

(c) Què afegiries per a garantir que en tots els bucles les iteracions es reparteixen de 2 en 2 entre els fils?

Qüestió 3–10

La següent funció processa una sèrie de transferències bancàries. Cada transferència té un compte origen, un compte destí i una quantitat de diners que es mou del compte origen al compte destí. La funció actualitza la quantitat de diners de cada compte (array `saldos`) i a més retorna la quantitat màxima que es transfereix en una sola operació.

```
double transferencies(double saldos[], int origens[], int destins[],
       double quantitats[], int n)
{
   int i, i1, i2;
   double diners, maxtransf=0;

   for (i=0; i<n; i++) {
      /* Processar transferència i: La quantitat transferida és quantitats[i],
       * que es mou del compte origens[i] al compte destins[i]. S'actualizen
       * els saldos de ambdós comptes i la quantitat màxima */
      i1 = origens[i];
      i2 = destins[i];
      diners = quantitats[i];
      saldos[i1] -= diners;
      saldos[i2] += diners;
      if (diners>maxtransf) maxtransf = diners;
   }
```

```
   return maxtransf;
}
```

(a) Paral·leliza la funció de forma eficient mitjançant OpenMP.

(b) Modifica la solució de l'apartat anterior perquè s'imprimisca l'índex de la transferència amb més diners.

Qüestió 3–11

Siga la següent funció:

```
double funcio(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) Paral·lelitza el codi anterior usant per a açò OpenMP. Explica les decisions que prengues. Es valoraran més aquelles solucions que siguen més eficients.

(b) Calcula el cost seqüencial, el cost paral·lel, l'speedup i l'eficiència que podran obtenir-se amb p processadors suposant que N és divisible entre p.

Qüestió 3–12

La següent funció proporciona totes les posicions de fila i columna en les quals es troba repetit el valor màxim d'una matriu:

```
int funcio(double A[N][N],double posicions[][2])
{
  int i,j,k=0;
  double maxim;
  /* Calculem el màxim */
  maxim = A[0][0];
  for (i=0;i<N;i++) {
    for (j=0;j<N;j++) {
      if (A[i][j]>maxim) maxim = A[i][j];
    }
  }
```

```
  /* Una vegada localitzat el màxim, busquem les seues posicions */
  for (i=0;i<N;i++) {
    for (j=0;j<N;j++) {
      if (A[i][j] == maxim) {
        posicions[k][0] = i;
        posicions[k][1] = j;
        k = k+1;
      }
    }
  }
  return k;
}
```

(a) Paral·lelitza la funció de forma eficient mitjançant OpenMP, fent ús d'una sola regió paral·lela.

(b) Modifica el codi de l'apartat anterior perquè cada fil imprimisca per pantalla el seu identificador i la quantitat de valors màxims que ha trobat i ha incorporat a la matriu `posicions`.

Qüestió 3–13

Es disposa d'una matriu `M` que emmagatzema dades sobre les actuacions dels `NJ` components d'un equip de bàsquet en diferents partits. Cadascuna de les `NA` files de la matriu correspon a l'actuació d'un jugador en un partit, emmagatzemant, en les seues 4 columnes, el dorsal del jugador (numeració consecutiva de 0 a `NJ`-1), el nombre de punts anotats pel jugador en el partit, el nombre de rebots aconseguits i el nombre de taps assolits. La valoració individual d'un jugador per cada partit es calcula d'aquesta manera:

$$\text{valoracio} = \text{punts} + 1.5 * \text{rebots} + 2 * \text{taps}$$

Paral·lelitza, mitjançant OpenMP i amb una única regió paral·lela, la següent funció encarregada d'obtenir i mostrar per pantalla el jugador que més punts ha anotat en un partit, a més de calcular la valoració mitjana de cada jugador de l'equip.

```
void valoracio(int M[][4], double valoracio_mitja[NJ]) {
  int i,jugador,punts,rebots,taps,max_punts=0,max_anotador;
  double suma_valoracio[NJ];
  int num_partits[NJ];
  ...
  for (i=0;i<NA;i++) {
    jugador = M[i][0];
    punts   = M[i][1];
    rebots  = M[i][2];
    taps    = M[i][3];
    suma_valoracio[jugador] += punts+1.5*rebots+2*taps;
    num_partits[jugador]++;
    if (punts>max_punts) {
      max_punts = punts;
      max_anotador = jugador;
    }
  }
  printf("Maxim anotador: jugador %d (%d punts)\n",max_anotador,max_punts);
  for (i=0;i<NJ;i++) {
    if (num_partits[i]==0)
      valoracio_mitja[i] = 0;
    else
      valoracio_mitja[i] = suma_valoracio[i]/num_partits[i];
```

```
  }
  ...
}
```

Qüestió 3–14

S'està celebrant un concurs de fotografia en el què els jutges atorguen punts a aquelles fotos que desitgen.

Es disposa d'una funció que rep els punts atorgats en les múltiples valoracions efectuades per tots els jutges, i un vector `totals` on s'acumularan eixos punts. Este vector `totals` ja està inicialitzat a zeros.

La funció calcula els punts totals per a cada foto, mostrant per pantalla les dues majors puntuacions atorgades a una foto en les valoracions. També calcula i mostra la puntuació final mitja de totes les fotos, així com el nombre de fotos que passen a la següent fase del concurs, que són les que reben un mínim de 20 punts.

Cada valoració `k` atorga una puntuació de `punts[k]` a la foto número `index[k]`. Lògicament, una mateixa foto pot rebre múltiples valoracions.

Paral·lelitza esta funció de forma eficient amb OpenMP, utilitzant una sola regió paral·lela.

```
/* nf = nombre de fotos, nv = nombre de valoracions */
void concurs(int nf, int totals[], int nv, int index[], int punts[])
{
  int k,i,p,t, passen=0, max1=-1,max2=-1, total=0;
  for (k = 0; k < nv; k++) {
    i = index[k]; p = punts[k];
    totals[i] += p;
    if (p > max2)
      if (p > max1) { max2 = max1; max1 = p; } else max2 = p;
  }
  printf("Les dues puntuacions més altes han sigut %d y %d.\n",max1,max2);
  for (k = 0; k < nf; k++) {
    t = totals[k];
    if (t >= 20) passen++;
    total += t;
  }
  printf("Puntuació mitja: %g. %d fotos passen a la següent fase.\n",
   (float)total/nf, passen);
}
```

Qüestió 3–15

La següent funció processa la facturació, a final del mes, de totes les cançons descarregades per un conjunt d'usuaris d'una tenda de música virtual. Per a cadascuna de les n descàrregues realitzades, s'emmagatzema l'identificador de l'usuari i el de la cançó descarregada, respectivament en els vectors `usuaris` i `cansons`. Cada cançó té un preu diferent, recollit en el vector `preus`. La funció mostra a més per pantalla l'identificador de la cançó que s'ha descarregat en més ocasions. Els vectors `ndescarregues` y `facturacio` estan inicialitzats a 0 abans d'invocar a la funció.

```
void facturacions(int n, int usuaris[], int cansons[], float preus[],
                  float facturacio[], int ndescarregues[])
{
  int i,u,c,millor_canso=0;
  float p;
  for (i=0;i<n;i++) {
     u = usuaris[i];
```

```
      c = cansons[i];
      p = preus[c];
      facturacio[u] += p;
      ndescarregues[c]++;
   }
   for (i=0;i<NC;i++) {
      if (ndescarregues[i]>ndescarregues[millor_canso])
         millor_canso = i;
   }
   printf("La cançó %d és la més descarregada\n",millor_canso);
}
```

(a) Paral·lelitza eficientment la funció anterior emprant una única regió paral·lela.

(b) Seria vàlid emprar la clàusula `nowait` en el primer dels bucles?

(c) Modifica el codi de la funció paral·lelitzada, de manera que cada fil mostre per pantalla el seu identificador i el nombre d'iteracions del primer bucle que ha processat.

Qüestió 3–16

Volem obtindre la distribució de les qualificacions obtingudes pels alumnes de CPA, calculant el nombre de suspensos, aprovats, notables, excel·lents i matrícules d'honor.

```
void histograma(int histo[], float notes[], int n) {
  int i, nota;
  float rnota;
  for (i=0;i<5;i++) histo[i] = 0;
  for (i=0;i<n;i++) {
    rnota = round(notes[i]*10)/10.0;
    if (rnota<5) nota = 0;        /* suspens */
    else
      if (rnota<7) nota = 1;      /* aprovat */
      else
        if (rnota<9) nota = 2;    /* notable */
        else
          if (rnota<10) nota = 3; /* excel·lent */
          else
            nota = 4;             /* matricula d'honor */
    histo[nota]++;
  }
}
```

(a) Paral·lelitzeu adequadament la funció `histograma` amb OpenMP.

(b) Modifica la funció `histograma` per a que mostre per pantalla el número de l'alumne amb la millor nota i la seua nota, i el valor de la pitjor nota (ambdues notes sense arrodonir).

Qüestió 3–17

La següent funció gestiona un nombre determinat de viatges, que han tingut lloc durant un període concret de temps, mitjançant el servei públic de bicicletes d'una ciutat. Per a cadascun dels viatges realitzats s'emmagatzemen els identificadors de les estacions origen i destinació, junt amb el temps (expressat en minuts) de duració. El vector `num_bicis` conté el nombre de bicicletes presents en cada estació. A més, la funció calcula entre quines estacions va tindre lloc el viatge més llarg i el més curt, junt amb el temps mitjà de duració de la totalitat dels viatges.

```
struct viatge {
   int estacio_origen;
```

```
   int estacio_desti;
   float temps_minuts;
};
void actualitza_bicis(struct viatge viatges[],int num_viatges,int num_bicis[]) {
   int i,origen,desti,ormax,ormin,destmax,destmin;
   float temps,tmax=0,tmin=9999999,tmitja=0;
   for (i=0;i<num_viatges;i++) {
      origen  = viatges[i].estacio_origen;
      desti = viatges[i].estacio_desti;
      temps  = viatges[i].temps_minuts;
      num_bicis[origen]--;
      num_bicis[desti]++;
      tmitja += temps;
      if (temps>tmax) {
         tmax=temps; ormax=origen; destmax=desti;
      }
      if (temps<tmin) {
         tmin=temps; ormin=origen; destmin=desti;
      }
   }
   tmitja /= num_viatges;
   printf("Temps mitjà entre viatges: %.2f minuts\n",tmitja);
   printf("Viatge més llarg (%.2f min.) estació %d a %d\n",tmax,ormax,destmax);
   printf("Viatge més curt (%.2f min.) estació %d a %d\n",tmin,ormin,destmin);
}
```

Paral·lelitzeu la funció mitjançant OpenMP de la forma més eficient possible.