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];
}
```

Solució: Hi ha una dependència de dades entre les diferents iteracions: incompleix la 1ª condició de Bernstein ($I_j \cap O_i \neq \emptyset$), doncs, per exemple, x[2] és una variable d'eixida en la iteració i=1 i una variable d'entrada en la iteració i=2. No és possible eliminar aquesta dependència de dades.

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

Solució: Hi ha una dependència de dades entre les diferents iteracions: incompleix la 3ª condició de Bernstein ($O_i \cap O_j \neq \emptyset$), doncs `x` és una variable d'eixida en totes les iteracions. En aquest cas sí és possible eliminar la dependència:

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

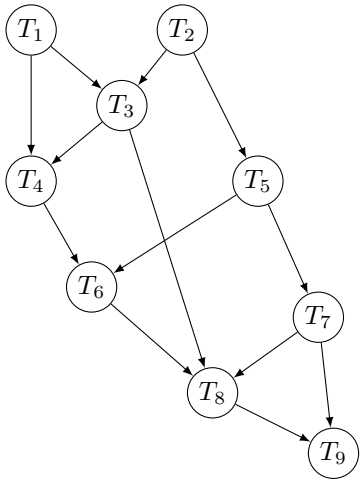
(c)
```
for (i=N-2;i>=0;i--) {
  x[i] = x[i] + y[i+1];
  y[i] = y[i] + z[i];
}
```

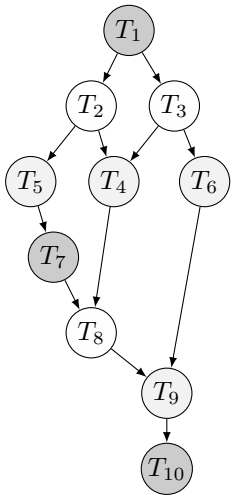
Solució: Hi ha una dependència de dades entre les diferents iteracions: incompleix la 1ª condició de Bernstein ($I_j \cap O_i \neq \emptyset$), doncs, per exemple, y[1] és una variable d'eixida en la iteració i=1 i d'entrada en la iteració i=0. En aquest cas sí és possible eliminar la dependència:

```
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];
```

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.

Solució: De tots els possibles camins entre un node inicial i un node final, el que major cost té (camí crític) és $T_1 - T_3 - T_4 - T_6 - T_8 - T_9$ (o de forma equivalent començant en T_2). La seua longitud és $L = 6$. El grau mitjà de concurrència és

$$M = \sum_{i=1}^{9} \frac{1}{6} = \frac{9}{6} = 1.5$$

(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.

Solució: En aquest cas, el camí crític és $T_1 - T_2 - T_5 - T_7 - T_8 - T_9 - T_{10}$ i la seua longitud és

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

El grau mitjà de concurrència és

$$M = \frac{3 \cdot \frac{1}{3}n^3 + 3 \cdot n^3 + 4 \cdot 2n^3}{7n^3} = \frac{12n^3}{7n^3} = 1.71$$

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.

Solució:

(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) Cost seqüencial: $t(N) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} 2 = 2N^2$ flops

Cost paral·lel: $t(N,p) = \sum_{i=0}^{d-1} \sum_{j=0}^{N-1} 2 = 2dN$ flops, on $d = \frac{N}{p}$

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

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

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).

Solució: El càlcul té dues fases. En la primera se sobreescriu cada fila de la matriu amb la fila següent multiplicada per 2. En la segona part es realitza la suma de tots els elements de la matriu. En la primera fase es realitza només una operació en el bucle més interior, i per tant el seu cost és $\sum_{i=0}^{M-2}\sum_{j=0}^{N-1} 1 = \sum_{i=0}^{M-2} N = (M-1)N \approx MN$. [L'aproximació la fem en sentit asimptòtic, és a dir, suposant que tant N com M són suficientment grans.] La segona fase té un cost similar, llevat que el bucle `i` fa una iteració més: MN.
Cost seqüencial: $t_1 = 2MN$ flops

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

Solució: Plantegem una paral·lelització amb dues regions paral·leles, una per cada fase, ja que la segona fase no pot començar fins que haja acabat la primera. En la primera fase existeixen dependències de dades en l'índex `i`, que poden resoldre's intercanviant els bucles i paral·lelitzant el bucle `j` (també es podria paral·lelitzar el bucle `j` sense intercanviar els bucles, però açò seria més ineficient). La segona fase requereix una reducció sobre la variable `suma`. En ambdues fases tant `i` com `j` han de ser variables privades.

```
double funcio(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 podrà obtenir-se amb p processadors suposant M i N múltiples exactes de p.

Solució: En paral·lel, el cost de la primera fase seria $\frac{N}{p}M$ i el de la segona fase $\frac{M}{p}N$ (en aquest segon terme hem menyspreat el cost associat a la reducció que fa internament OpenMP sobre la variable `suma`).
El temps paral·lel seria: $t_p = \dfrac{2MN}{p}$ flops
El speedup serà per tant: $S_p = \dfrac{t_1}{t_p} = \dfrac{2MN}{2MN/p} = p$

(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).

Solució: En aquest cas, el temps paral·lel és $t_p = MN + \frac{MN}{p}$ flops, i per tant el speedup serà

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

el límit del qual quan $p \to \infty$ és 2. És a dir, el speedup mai podrà ser major que dos encara que usem molts processadors.

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?

Solució: Primer bucle:

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

Segon bucle:

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

Tercer bucle:

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

La forma més eficient consisteix en paral·lelitzar el bucle més extern, doncs es produeix una menor sobrecàrrega deguda a l'activació i desactivació de fils, i també es redueixen els temps d'espera deguts a la sincronització implícita al final de la directiva.

(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.

Solució: Cost seqüencial: $t_1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \left(1 + \sum_{k=0}^{n-1} 2\right) \approx 2n^3$ flops.

Cost paral·lel: $t_n = 2n^3/n = 2n^2$ flops.

Speedup: $S_n = t_1/t_n = n$.

(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.

Solució: Es declaren dues variables enteres, `iter` i `tid` (`iter` ha d'estar inicialitzada a 0 i `tid` ha d'aparèixer com a privada en la clàusula `parallel`, amb `private(tid)`), i s'afigen les següents línies:
Entre el primer i el segon `for`:

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

Després dels bucles niats (abans del `return`):

```
printf("Nombre d'iteracions realitzades pel fil 0 = %d\n",iter);
```

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 dinámica 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.

Solució:

```
#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 fil que ha sumat l'últim nombre és %d\n",tid);
  printf("La suma de los primers %u números naturals és %u\n",n,s);
}
```

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.

Solució:

```
#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 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.

Solució:

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

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

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

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.

Solució: El bucle més exterior no pot paral·lelitzar-se perquè hi ha una dependència de cada iteració amb l'anterior, en utilitzar com `x` el valor obtingut en `x2` en la iteració anterior. En el bucle `i` la variable `s` acumula el valor del producte de la fila pel vector però es calcula de forma completa en cada iteració, per la qual cosa haurà de ser privada (igual que la variable del bucle més interior, `j`). No és aquest el cas de la variable `q`, el valor de la qual és el resultat de multiplicar els valors de les diferents iteracions, requerint una reducció. Com hi ha una dependència entre els

dos bucles `for` de `i`, no és necessari utilitzar una única regió paral·lela ja que no podem utilitzar la clàusula `nowait`. La variable `max` no necessita protecció perquè no hi ha condicions de carrera, ja que totes les iteracions del bucle més exterior són seqüencials.

```
#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;
    #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 cost teòric (en flops) que tindria una iteració del bucle `k` del codi seqüencial.

Solució:

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

(c) 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.

Solució:

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

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

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

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`.

Solució: Bastaria amb afegir la següent directiva immediatament abans del bucle:

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

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

Solució: Bastaria amb afegir la següent directiva immediatament abans del bucle:

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

(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.

Solució: El temps d'una iteració del bucle `i` seria:

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

i el temps de la funció completa és la suma del temps de totes les iteracions del bucle, o siga:

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

(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.

Solució: En l'apartat anterior vegem que el cost d'una iteració del bucle `i` és major quant major siga el valor de `i`. Si utilitzem la planificació `schedule(static)` no hi haurà bon equilibri de càrrega, perquè el bloc d'iteracions més costoses (amb major valor de `i`) li tocaran a un mateix fil.

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.

Solució: És convenient paral·lelitzar el bucle més extern. Bastaria amb afegir la següent directiva immediatament abans del bucle `i`:

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

Les distintes iteracions del bucle `i` tenen cost diferent, donat que el recorregut del bucle `k` depén del valor de `i`. Per tant, és d'esperar que la planificació sí afecte. Com per a dos valors consecutius del valor de `i` el valor de `k` varia en 1, podrien ser adequades les planificacions `static` o `dynamic` amb el valor de *chunk* igual a 1, és a dir, `schedule(static,1)` o `schedule(dynamic,1)`.

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

Solució:

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

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.

Solució:
Just abans del bucle `i` col·locaríem la directiva:

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

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

Solució:

```
...
  aux = 0;
  #pragma omp parallel
  {
    #pragma omp for private(x) reduction(+:aux) nowait
    for (j=0; j<N; j++) {
      ...
    }
    #pragma omp for private(y)
    for (j=i; j<N; j++) {
      ...
    }
  } /* fi del parallel */
  vs[i] = aux;
...
```

(c) Calculeu el cost seqüencial del codi original.

Solució:

$$t(N) = \sum_{i=0}^{N-1} \left(\sum_{j=0}^{N-1} 3 + \sum_{j=i}^{N-1} 1 + 1 \right) \approx \sum_{i=0}^{N-1} (3N + N - i) = 4N^2 - \sum_{i=0}^{N-1} i \approx 4N^2 - \frac{N^2}{2} = \frac{7N^2}{2} \text{ flops}$$

(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.

Solució:

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

Si $p = N$:

$$t(N,p) = 3N + \frac{N^2}{2} \approx \frac{N^2}{2} \text{ flops}$$

Per tant el speedup i l'eficiència seran:

$$S(N,p) = \frac{\frac{7N^2}{2}}{\frac{N^2}{2}} = 7; \quad E(N,p) = \frac{7}{N}$$

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;
}
```

Solució: Declarem com `volatile` la variable `trobat` per a garantir que en el moment en què un fil modifique el seu valor la resta de fils veuran aquest canvi.

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

Solució:

```
double norma(double v[], int n)
{
   int i, i_fil, p;
   double r=0;

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

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

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

Solució:

```
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 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.

Solució: Cost de l'algorisme seqüencial (el cost de l'arrel quadrada és menyspreable enfront del cost del bucle `i`):

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

Cost de l'algorisme paral·lel: com cada iteració del bucle `i` costa 2 flops i cada fil realitza n/p iteracions, el cost és de $t(n,p) = 2n/p$ flops. El speedup és de $S(n,p) = 2n/(2n/p) = p$.

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.

Solució:

```
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("Fil %d: %d iteracions processades\n",
                omp_get_thread_num(), cont);
   }
}
```

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.

Solució: El càlcul té dues fases. En la primera es calcula la suma dels quadrats dels elements de la matriu. En la segona part s'escalen els elements de la matriu. Amb dues regions paral·leles es garanteix que la segona fase no comença abans que haja acabat la primera, en cas contrari el càlcul podria ser incorrecte. La primera fase requereix una reducció sobre la variable `suma`. En ambdós casos la variable `j` deu ser privada. La variable `factor` és compartida i la calcula el fil principal (fora de les regions paral·leles).

```
void normalitza(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) 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.

Solució:

```
void normalitza(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 clàusula `nowait` s'utilitzaria per a eliminar la barrera implícita al final de la primera directiva `for`. No obstant, en aquest caso no es pot utilitzar perquè la barrera és necessària, ja que seria incorrecte que un dels fils passara a executar el següent `for` mentre altres fils estan encara en l'anterior (la segona fase modifica els elements de la matriu mentre que la primera fase els llig). Per altra banda, la clàusula `nowait` en combinació amb `reduction` podria fer que algun dels fils calculara un valor incorrecte de `factor`, en cas de no haver-se completat la reducció.

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.

Solució:

```
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;
      }
```

Com entre els dos primers bucles `for` no hi ha dependència de dades, és convenient utilitzar la clàusula `nowait` en el primer bucle paral·lel `for`, per a així evitar la sincronització que es produeix en la finalització de la directiva `for`.

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

Solució:

Temps seqüencial:

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

suposant M i N suficientment grans (cost asimptòtic).

Temps paral·lel:

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

suposant M i N suficientment grans (cost asimptòtic).

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

Solució: Speedup:

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

Eficiència:

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

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;
```

Solució: L'única cosa a tindre en compte és que la variable `a` deu 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;
  }
}
```

Qüestió 2–7

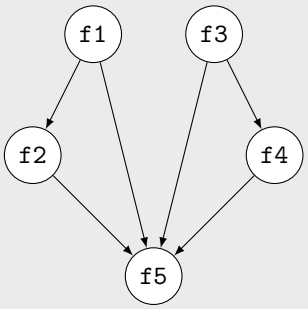
Donada la següent funció:

```
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.

Solució: El graf de dependències es mostra a continuació:



Les dependències de `f5` respecte a les altres tasques són dependències de flux (les seues entrades són generades per altres tasques). Les dependències de `f2` amb `f1` i `f4` amb `f3` són anti-dependències (no és que necessiten l'eixida d'altres tasques, sinó que modifiquen l'entrada de tasques prèvies i per tant no han de realitzar-se fins que no han acabat aquelles tasques prèvies). També existeix una dependència d'eixida entre `f5` i `f1` ja que ambdues modifiquen la mateixa dada d'eixida (la `a`).

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

Solució:

```
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) 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?

Solució: Per comoditat, assumim que cada funció tarda 1 unitat de temps. El temps seqüencial és

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

Per a aconseguir un speedup gran convé reduir el màxim possible el temps d'execució. Açò ens porta a usar tants processadors com tasques paral·leles es poden executar concurrentment. Donat el programa paral·lel i el graf de dependències, n'hi ha prou amb 2 processadors (fils). En eixe cas, el cost seria (`f1` i `f2` es fan en paral·lel a `f3` i `f4`):

$$t_p = 2 + 1 = 3$$

$$Sp = t_1/t_p = 5/3 = 1.67$$

Per a millorar l'speedup, es podrien eliminar les anti-dependències replicant les dades que van a ser modificades. Caldria veure si el cost de la còpia i l'espai extra necessari compensen el guany de velocitat que s'obtindria. Fent açò, es copiarien `b` i `d` abans de començar a treballar. `f1` i `f3` treballarien amb les copies i `f2` i `f4` amb les dades reals. Amb açò es podrien fer `f1`,`f2`,`f3`,`f4` a la volta, amb la qual cosa es tindria (sense tintre en compte el sobrecost de les copies):

$$t_p = 1 + 1 = 2 \qquad \text{(treballant amb 4 processadors)}$$

$$S_p = 5/2 = 2.5$$

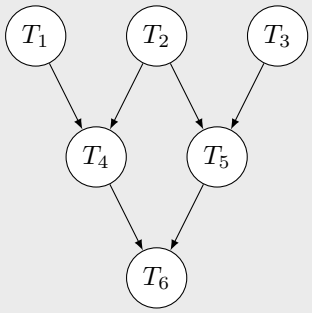
Qüestió 2–8

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.

Solució:



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

Solució:

```
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);    /* Tasca T1 */
         #pragma omp section
         T2(y,n);    /* Tasca T2 */
         #pragma omp section
         T3(z,n);    /* Tasca T3 */
      }

      #pragma omp sections
      {
         #pragma omp section
         /* 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;
         }

         #pragma omp section
         /* 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;
    }
```

(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.

Solució: El cost a priori de T_4 és 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 cost de T_5 és igual al de T_4, i el de T_6 és de $2n+1$ flops.
Per tant, el cost seqüencial és de:

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

Si el nombre de fils, p, és al menys 3, el cost de l'algoritme paral·lel serà de $t(n,p)=2n^2+3n^2+2n\approx 5n^2$ flops, i l'speedup serà:

$$S(n,p)=\frac{12n^2}{5n^2}=2.4$$

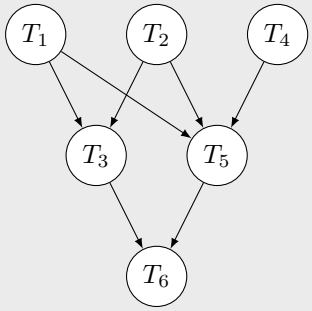
Qüestió 2–9

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.

Solució: La tasca T_5 modifica `x`, i per tant té una anti-dependència respecte de T_1, T_2 i T_4.



(b) Paral·lelitza el codi mitjançant OpenMP.

Solució:

```
#pragma omp parallel
{
   #pragma omp sections
   {
      #pragma omp section
      minx = minim(x,n);        /* T1 */
      #pragma omp section
      maxx = maxim(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 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.

Solució: Longitud del camí crític: $L = 2n + n + n = 4n$ flops

Grau mitjà de concurrència: $\frac{7n}{4n} = 7/4 = 1.75$

Per a obtindre l'speedup i l'eficiència, hi ha que tindre en compte que el temps d'execució seqüencial és $t(n) = 7n$, i el temps d'execució paral·lel amb 5 processadors és $t(n, 5) = 2n + n + n = 4n$ flops. Per tant:

$$S(n,p) = \frac{7n}{4n} = 1.75$$

$$E(n,p) = \frac{1.75}{5} = 0.35$$

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);
```

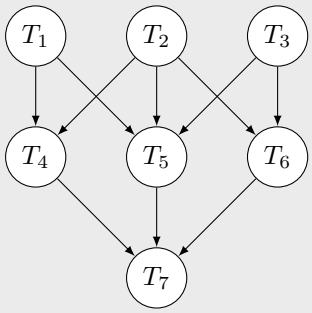
(a) Paral·lelitza el codi de forma eficient a nivell de bucles.

Solució: El bucle de `fun1` no es pot paral·lelitzar a causa de les dependències entre les diferents iteracions, per la qual cosa es considera únicament la funció `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) Dibuixa el graf de dependències de tasques, segons la numeració de tasques indicada en el codi.

Solució:



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

Solució: El codi paral·lel del programa principal és el següent (la resta queda invariant):

```
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) 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.

Solució: El temps seqüencial és:

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

El temps paral·lel i l'speed-up per al primer algoritme considerant 3 processadors és:

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

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

El temps paral·lel i l'speed-up per al segon algoritme considerant 3 processadors és:

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

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

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);
```

Solució: La paral·lelització a nivell de bucles no és possible en aquest cas ja que el nombre d'iteracions no és conegut a priori. Per tant, s'aborda una paral·lelització per tasques, situant cada bucle (incloent la inicialització) en una secció concurrent. Perquè el resultat siga correcte, deurem a més tenir en compte que les variables `x`, `i`, `z` han de tenir un àmbit privat, ja que les modifiquen els tres bucles però el seu valor no és compartit entre els bucles (s'inicialitzen al principi de cada bucle). Finalment, la variable `w` ha de tenir com a valor final el valor acumulat de totes les seccions concurrents, per al que s'utilitzarà una clàusula `reduction`.

```
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 */

#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);
```

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.

Solució: En ambdues funcions, la forma més senzilla i eficient de realitzar la paral·lelització a nivell de bucles és paral·lelitzar mitjançant la directiva `parallel for` el bucle més extern. D'aquesta manera, les iteracions d'aquest bucle es reparteixen entre els diferents fils de manera que cada fil s'encarrega del càlcul associat a un conjunt determinat de files de la matriu resultat. Hem de definir, addicionalment, l'abast de les variables, de manera que les variables `j`, `k` i `summa` han de ser privades per a cada fil.

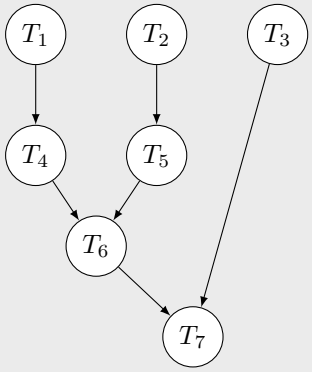
```
void matmult(double A[N][N],
       double B[N][N],double C[N][N])
{
  int i,j,k;
  double suma;
  #pragma omp parallel for private(j,k,suma)
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      suma = 0.0;
      for (k=0; k<N; k++) {
        suma = suma + A[i][k]*B[k][j];
      }
      C[i][j] = suma;
    }
  }
}
```

```
void simetritza(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) 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.

Solució: Les 3 primeres tasques no presenten dependències entre si, per la qual cosa poden executar-se concurrentment. Existeix per contra una dependència entre les tasques T_1 i T_4, i entre les tasques T_5 i T_2 (deguda a les variables `C1` i `C2`, respectivament). Les tasques T_4 i T_5 també poden executar-se simultàniament al no existir dependència alguna entre elles. Les dades d'entrada de la tasca T_6 (`C1` i `C2`) són les dades d'eixida de T_4 i T_5, existint per tant una dependència de flux entre elles, i de forma similar entre T_3 i T_6 i la tasca T_7. El graf de dependències és el següent:



El grau màxim de concurrència es 3, ja que com a molt hi haurà 3 tasques executant-se concurrentment.

El cost en flops de `matmult` (c_m) i de `simetritza` (c_s) és:

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

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

Per tant, cinc de les tasques tenen un cost de $2N^3$, mentre que T_4 i T_5 tenen un cost molt menor ($N^2/2$). El cost total acumulat de totes les tasques és $C = 10N^3 + N^2$ (cost seqüencial).

El camí crític és T_1–T_4–T_6–T_7 (o, equivalentment, T_2–T_5–T_6–T_7), el cost del qual és

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

El grau mitjà de concurrència és

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

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

Solució: Podem agrupar les tasques T_1 amb T_4 i T_2 amb T_5, ja que no existeixen dependències creuades. Llavors, una possible solució seria fer una primera part amb dues seccions, i posteriorment executar T_3 i T_6 en paral·lel. Finalment quedaria executar la tasca T_7 després de la finalització de les anteriors. Aquesta aproximació només requereix dos fils concurrents. També podrien executar-se els dos primers grups amb T_3 en paral·lel i executar seqüencialment T_6 i T_7. Incloem la primera solució. Per a açò, dins d'una única regió paral·lela usem dues vegades la directiva `sections`

(amb dues seccions concurrents cada vegada) per a implementar mitjançant barreres implícites les dependències de la tasca T_6 amb les tasques T_4 i T_5. Amb aquesta implementació, totes les variables seran compartides, ja que no existiran tasques que s'estiguen executant simultàniament i que modifiquen les mateixes variables.

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

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.

Solució:

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

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

Solució: El bucle que realitza la suma prefixa no es pot paral·lelitzar, donat que existeixen dependències de dades entre les distints iteracions. Existeixen esquemes algorítmics (prou complicats) per a calcular la suma prefixa en paral·lel. No obstant això, en esta ocasió hem decidit realitzar esta fase de forma seqüencial, donat que té un cost lineal, mentres que les altres dues parts tenen un cost quadràtic, per la qual cosa la penalització per no paral·lelitzar esta operació serà xicoteta. Per a fer la suma prefixa de forma seqüencial és necessària una directiva `single`, perquè si tots els fils executaren eixa part hi hauria condicions 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 files */
      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 prefixa */
      s[i] += s[i-1];
    }
    #pragma omp for private(i)
    for (j=0; j<N; j++) {     /* escalat de columnes */
      for (i=0; i<N; i++) {
        A[i][j] *= s[j];
      }
    }
  }
}
```

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

Solució: El temps paral·lel seria:

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

L'speedup serà per tant:

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

S'obté l'speedup ideal, encara que el valor obtingut hauria sigut menor si en les expressions de t_1 i t_p s'hagueren mantingut els termes d'ordre lineal.

Qüestió 2–14

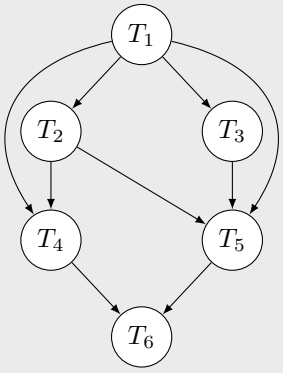
Donada la següent funció:

```
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.

Solució: La tasca T_1 modifica els seus dos arguments, per la qual cosa les tasques T_2 i T_3 tenen una dependència amb la tasca T_1. La tasca T_4 té una dependència amb T_1 i T_2, mentre que la tasca T_5 depèn de T_1, T_2 i T_3. La tasca T_6 depèn de T_4 i T_5. . El graf de dependències seria així:



El grau màxim de concurrència és 2 (per exemple, T_2 y T_3 es poden fer a la vegada).

Tenint en compte el cost de cada tasca, el camí crític pot passar indistintament per T_2 o T_3 (tenen el mateix cost), però ha de passar per T_5, donat que T_5 té major cost que T_4. Hi ha dos possibles camins crítics: el T_1, T_2, T_5, T_6 i el T_1, T_3, T_5, T_6. La longitud de qualsevol dels dos és:

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

El grau mitjà de concurrència seria:

$$M = \frac{\sum_{i=1}^{6} c_i}{L} = \frac{7n^2 + 1}{5n^2 + 1} \approx 1.4$$

(b) Paral·lelitza la funció amb OpenMP.

Solució: Utilitzem la directiva `sections` en tractar-se d'una aproximació basada en paral·lelisme de tasques. La dependència existent entre les tasques T_2 i T_5 fa que no es puguen agrupar les tasques $T_2 - T_4$ i les tasques $T_3 - T_5$, sent necessari que es produïsca una sincronització entre el primer bloc de tasques concurrents (T_2 i T_3) i el segon (T_4 i T_5). Utilitzarem per tant dues regions paral·leles amb `sections` separades. La tasca T_6 es realitzarà de forma seqüencial al final de les regions paral·leles, igual que T_1 al principi del programa.

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

  omplir(A,B);               /* T1 */
  #pragma omp parallel sections
  {
    #pragma omp section
    a = calculs(A);          /* T2 */
    #pragma omp section
    b = calculs(B);          /* T3 */
  }
  #pragma omp parallel sections
  {
    #pragma omp section
    x = suma_menors(B,a);    /* T4 */
    #pragma omp section
    y = suma_en_rang(B,a,b); /* T5 */
  }
  z = x + y;                  /* T6 */

  return z;
}
```

(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.

Solució: Si s'executa amb 2 o més fils, com en cada `sections` del codi hi ha un màxim de 2 `section`, ambdues `section` es podran fer en paral·lel. Per tant, per al temps paral·lel cal agafar el temps màxim de les diferents tasques presents en cada `sections`.

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

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

$$S_p = \frac{t_1}{t_3} \approx 1.4$$

$$E = \frac{Sp}{p} = \frac{Sp}{3} = 46.67\%$$

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).

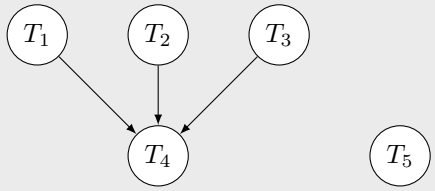
```
typedef double image[N][M];

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).

Solució: El graf de dependències es mostra a continuació:



El grau màxim de concurrència seria 4, donat que T_1, T_2, T_3 i T_5 es poden executar en paral·lel. Per a obtindre el grau mitjà de concurrència, cal determinar el cost en flops de les funcions `difer` i `suma`:

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

El camí crític és $T_1 \longrightarrow T_4$ (o qualsevol dels altres dos camins del graf que acaben en T_4, que tenen el mateix cost), per la qual cosa la longitud del camí crític és:

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

Per tant,

$$\text{Grau mitjà de concurrència} = \frac{N \cdot M + N \cdot M + N \cdot M + 2 \cdot N \cdot M + N \cdot M}{L} = \frac{6}{3} = 2$$

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

Solució: Realitzem la paral·lelització mitjançant seccions. La tasca T_5 es pot executar en paral·lel amb qualsevol de les altres tasques. En el codi paral·lel l'hem situada en una construcció `sections` junt amb T_4, donat que eixa seria la solució amb major concurrència en el cas de tindre 3 fils.

```
void processa(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 */
      }
    }
  }
```

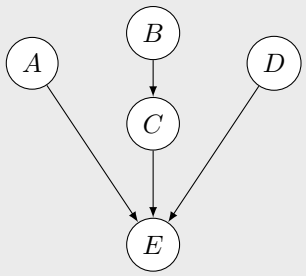
Qüestió 2–16

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.

Solució: Les tasques A, B i D no presenten cap dependència entre si i poden executar-se concurrentment. Per contra, la tasca C té, entre les seues dades d'entrada, a la variable `aux` generada per la tasca B. Existeix per tant una dependència de flux entre elles. D'igual manera, la tasca E té una dependència de flux amb les tasques A, C i D, ja que les 3 variables que suma s'han obtingut a partir de l'execució de les 3 tasques citades.



Grau màxim de concurrència: 3 (per exemple, es poden fer a la vegada les tasques A, B i D).
El camí crític és $A \to E$ i la seua longitud és $3n^2 + 2$ flops.

El grau mitjà de concurrència és $\frac{3n^2+n^2+n^2+2n^2+2}{L} = \frac{7n^2+2}{3n^2+2} \approx 7/3 = 2.33$.

(b) Paral·lelitza-la amb OpenMP.

Solució: Recorrem a la directiva `sections` emprant una única regió paral·lela i 3 seccions concurrents, després d'agrupar les tasques B i C (el cost agregat d'aquestes tasques és menor o igual al de les altres tasques concurrents A i D). La tasca E no forma part de la regió paral·lela i s'executarà pel fil principal quan hagen acabat les tasques anteriors.

```
double calculs_matricials(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 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.

Solució: $t_1 = 3n^2 + n^2 + n^2 + 2n^2 + 2 = 7n^2 + 2$ flops

Si executem el codi paral·lel amb 2 fils, el repartiment de càrrega més equilibrat correspon al cas en què un fil fa la tasca A i l'altre executa les tasques B, C i D. Finalment, el fil principal executa la tasca E. Per tant,

$t_2 = \max(3n^2, n^2 + n^2 + 2n^2) + 2 = \max(3n^2, 4n^2) + 2 = 4n^2 + 2$ 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 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.

Solució: Per a calcular el temps d'execució, emprem la funció `omp_get_wtime` abans i després de la regió paral·lela corresponent a l'apartat anterior, restant els valors de temps proporcionats per a obtenir el temps transcorregut. Fora de la regió paral·lela, el fil principal mostrarà aquest temps per pantalla. Per a obtenir el nombre de fils, creem una regió paral·lela addicional on invoquem la funció `omp_get_num_threads`, mostrant el valor obtingut, una vegada conclosa la mateixa, pel fil principal.

```
#include <omp.h>
double calculs_matricials(double mat[n][n])
{
  double x,y,z,aux,total,t1,t2,t;
```

```
    int num_fils;
    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_fils = omp_get_num_threads();
    printf("Temps amb %d fils: %.2f segons\n",num_fils,t);
    return total;
  }
```

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.

Solució: Només caldria col·locar la següent directiva abans del bucle `i`, de manera que les variables `i`, `j`, `j2` i `a` siguen privades. Com tots els fils contribueixen parcialment a la variable `x`, hem d'emprar una reducció per a acumular el valor obtingut per cada fil.

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

(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).

Solució: La manera més eficient consisteix a emprar una única regió paral·lela que abaste els dos bucles `j` i `j2`. De nou hem d'emprar una reducció per a obtenir el valor correcte de les variables `x` i `y`, ja que tots els fils participen en el seu càlcul.
L'ús de la clàusula `nowait` evita la barrera implícita entre els dos bucles, de manera que els fils que ja han finalitzat la seua labor en el bucle `j` poden començar amb el bucle `j2`. Açò és així perqué que no hi ha cap dependència entre les variables d'aquests 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 cost (temps d'execució) del codi seqüencial.

Solució:

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

(d) 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.

Solució:

- Bucle `i`: les distintes iteracions del bucle `i` tenen cost diferent, perquè el recorregut del bucle `j2` depén del valor de `i`. Per tant, és d'esperar que la planificació afecte. Serien adequades planificacions *static* amb *chunk* 1, *dynamic* o *guided*.
- Bucle `j`: totes les iteracions del bucle tenen el mateix cost (1 flop), per la qual cosa en principi la planificació no afectaria a les prestacions (és possible que una planificació dinàmica funcione pitjor, per suposar certa sobrecàrrega).
- Bucle `j2`: ocorre el mateix que en el bucle `j` (en este cas cada iteració són 2 flops).

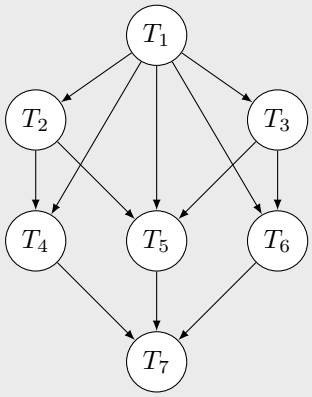
Qüestió 2–18

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.

Solució: A continuació es mostra el graf de dependències. Totes les dependències són de flux, excepte l'antidependència entre T_5 i les tasques T_2 i T_3 (T_5 modifica les variables `x` i `y`).



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

Solució: La funció paral·lelitzada seria la següent:

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

```
        }
      }
      escriu_resultats(c,d,e);   /* Tasca 7 */
    }
```

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

Solució: El temps seqüencial és:

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

i el temps en paral·lel amb 3 processadors:

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

A partir d'eixos valors, l'increment de velocitat valdrà:

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

i l'eficiència:

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

(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.

Solució: Un dels camins crítics és $T_1 - T_2 - T_5 - T_7$ i la seua longitud és:

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

A partir de la longitud del camí crític, el grau mitjà de concurrència és:

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

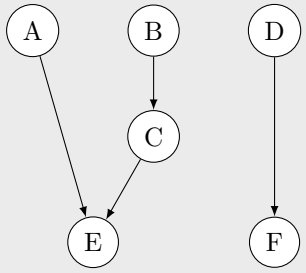
Qüestió 2–19

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.

Solució:



Camí crític: A⟶E
Longitud del camí crític: $L = 2n^2 + n^2 = 3n^2$ flops
Grau mitjà de concurrència: $M = \frac{2n^2+n+n^2+2n^2+n^2+3n}{3n^2} \approxeq \frac{6n^2}{3n^2} = 2$

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

Solució:

```
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) 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.

Solució: Les 3 seccions de la primera construcció `sections` s'haurien de repartir entre 2 fils. El millor cas seria que les tasques A i D es feren en fils diferents, i les tasques B i C en un fil qualsevol. Per exemple, un fil podria fer les tasques A, B i C (amb un cost de $2n^2 + n^2 + n \approxeq 3n^2$ flops) i l'altre fil la tasca D ($2n^2$ flops). Després, un fil executaria E (n^2 flops) mentres l'altre executa F ($3n$ flops). Per tant:

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

Tenint en compte que el cost seqüencial és la suma del cost de totes les tasques, o siga $6n^2$ flops:

$$S_2 = \frac{6n^2}{4n^2} = \frac{6}{4} = 1.5$$

$$E_2 = \frac{1.5}{2} = 0.75$$

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.

Solució:

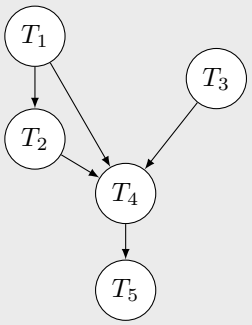
$$t_{\text{tasca4}} = 3$$

$$t_{\text{tasca5}}(n) = \sum_{i=0}^{n-1} 6 = 6n$$

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

(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.

Solució:



El màxim nombre de tasques que poden executar-se concurrentment és 2 (tasques 1 i 3 o tasques 2 i 3), així que el grau màxim de concurrència és 2.

El camí crític és el de major longitud que va d'un node inicial a un node final. En este cas serà el format per les tasques 3, 4 i 5 i la seua longitud és $L = 4n + 3 + 6n \approx 10n$.

El grau mitjà de concurrència es calcula com

$$M = \frac{\sum_{i=1}^{6} t_{\text{tasca i}}}{L} = \frac{12n+3}{10n+3} \approx \frac{12}{10} = 1.2$$

(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.

Solució:

```
int exercici(double v[n],double x)
{
  int i,j,k=0;
  double a,b,c;
  #pragma omp parallel
  {
    #pragma omp sections
    {
      #pragma omp section
      {
        a = tasca1(v,x);
        b = tasca2(v,a);
      }
      #pragma omp section
      c = tasca3(v,x);
    }
    #pragma omp single
    x = x + a + b + c;   /* tasca 4 */
    #pragma omp for private(j) reduction(+:k)
    for (i=0; i<n; i++) {   /* tasca 5 */
      j = f(v[i],x);
      if (j>0 && j<4) k++;
    }
  }
  return k;
}
```

(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.

Solució:

$$t_6(n) = \max\{t_{\text{tasca1}} + t_{\text{tasca2}}, t_{\text{tasca3}}\} + t_{\text{tasca4}} + \sum_{i=0}^{n/6-1} 6 = 4n + 3 + \frac{6n}{6} \approx 5n \text{ flops}$$

$$S_6 = \frac{t(n)}{t_6(n)} = \frac{12n}{5n} = \frac{12}{5} = 2.4$$

$$E_6 = \frac{S_6}{p} = \frac{2.4}{6} = 0.4 \rightarrow 40\%$$

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;
    }
  }
}
```

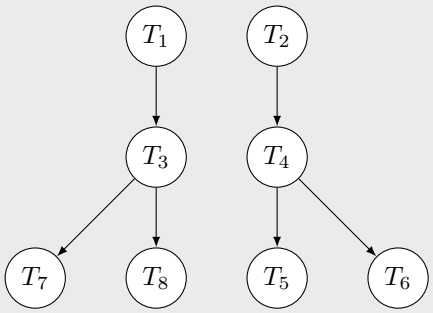
```
void normalize(double A[N][N]) {
  int i,j;
  double sum=0.0,factor;
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      sum += A[i][j]*A[i][j];
    }
  }
  factor = 1.0/sqrt(sum);
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      A[i][j] *= factor;
    }
  }
}
```

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.

Solució: El graf de dependències és el següent:



El cost en flops de `matmult` (c_m) i de `normalize` (c_n) és:

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

$$c_n = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} 2 + 6 + \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} 1 = 2N^2 + 6 + N^2 \approx 3N^2.$$

Per tant, sis de les tasques tenen un cost de $2N^3$, mentres que T_3 i T_4 tenen un cost molt menor ($3N^2$). El cost total acumulat de totes les tasques és $C = 12N^3 + 6N^2$ (cost seqüencial).

El camí crític és per exemple T_1–T_3–T_8 (hi ha altres camins amb el mateix cost), el cost del qual és

$$L = 2N^3 + 3N^2 + 2N^3 = 4N^3 + 3N^2.$$

El grau mitjà de concurrència és

$$M = \frac{C}{L} = \frac{12N^3 + 6N^2}{4N^3 + 3N^2} \approx \frac{12}{4} = 3.$$

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

Solució: Podem agrupar les tasques T_1 i T_3, per un costat, i T_2 i T_4, per altre, donat que no existeixen dependències creuades. Per tant, la paral·lelització es pot fer amb una primera part amb dos seccions, i posteriorment executar les quatre tasques restants en paral·lel.

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    {
      matmult(A,B,R1);    /* T1 */
      normalize(R1);      /* T3 */
    }
    #pragma omp section
    {
      matmult(C,D,R2);    /* T2 */
      normalize(R2);      /* T4 */
    }
  }
  #pragma omp sections
  {
    #pragma omp section
    matmult(A,R2,M1);   /* T5 */
    #pragma omp section
    matmult(B,R2,M2);   /* T6 */
    #pragma omp section
    matmult(C,R1,M3);   /* T7 */
    #pragma omp section
    matmult(D,R1,M4);   /* T8 */
  }
}
```

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.

Solució:

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

   #pragma omp parallel for private(maxim,j) reduction(+:suma)
   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;
}
```

(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*.

Solució: Es reparteixen les iteracions del bucle i entre els p fils disponibles, i cada iteració realitza els dos següents bucles complets, amb un cost de

$$\sum_{i=0}^{N/p-1}\left(\sum_{j=0}^{M-1}1+\sum_{j=0}^{M-1}3\right)=\sum_{i=0}^{N/p-1}4M=\frac{4NM}{p}$$

(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.

Solució:

```
double sumar(double A[N][M])
```

```
{
   double suma=0, maxim;
   int i, j, comptar;

   #pragma omp parallel private(maxim,j,comptar) reduction(+:suma)
   {
      comptar=0;
      #pragma omp for
      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];
               comptar++;
            }
         }
      }
      printf("El fil %d ha sumat %d elements.\n" ,
             omp_get_thread_num(), comptar);
   }
   return suma;
}
```

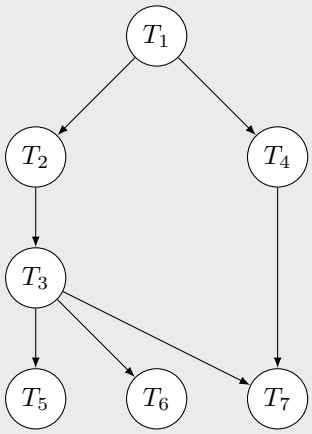
Qüestió 2–23

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.

Solució: A continuació es mostra el graf de dependències.



Entre T_2 i T_3 tenim una dependència de flux i una dependència d'eixida degut a la variable `c`. Entre T_4 i T_7 existeix una antidependència per part de la variable `b`.

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

Solució: De tots els camins possibles, el que es correspon amb el camí crític segueix la seqüència de tasques $T_1 - T_2 - T_3 - T_6$, amb longitud L=10+15+8+35=68 flops.

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

Solució:

```
double a,b,c,e,d,f;
T1(&a,&b);
#pragma omp parallel
{
   #pragma omp sections
   {
      #pragma omp section
      {
      c=T2(a);
      c=T3(c);
      }
      #pragma omp section
      d=T4(b);
   }
   #pragma omp sections
   {
      #pragma omp section
      e=T5(c);
      #pragma omp section
      f=T6(c);
      #pragma omp section
      b=T7(c);
   }
}
```

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

Solució: El cost paral·lel es pot obtindre a partir del graf de dependències, com la suma dels següents costos: T_1, el màxim entre $(T_2 + T_3)$ i T_4, i el màxim entre T_5, T_6 i T_7. Per tant, $t_p = 10 + 23 + 35 = 68$.

Obtenim el speedup com $S_p = \frac{t_1}{t_p} = \frac{148}{68} = 2.18$

Per últim, l'eficiència per a 4 fils la calculem com $E_p = \frac{S_p}{p} = \frac{2.18}{4} = 0.55$

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.

Solució: La paral·lelització del codi anterior consisteix en paral·lelitzar els bucles interns. El bucle `while` exterior no és paral·lelitzable, mentre que els interns sí ho són si ens fixem en les dependències de dades. Es tractaria simplement d'afegir la següent directiva

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

abans de cada bucle `for`. La variable `a` deu ser privada, les demés son compartides excepte la variable d'iteració `i` que és privada per defecte.

La variable `ordenat` és compartida i tots els fils accedeixen a ella per a escriure-la. Això deuria obligar a protegir-la en una secció crítica (o una operació de reducció). No obstant, en aquest cas no és necessari.

(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`.

Solució: La manera de comptar el nombre d'intercanvis consisteix a utilitzar una variable comptador (`int cont=0`). Aquesta variable ha de ser privada a cada fil, agregant el compte total a l'eixida del bucle mitjançant una operació de reducció. La directiva en els bucles seria la següent:

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

afegint en el cos del bucle (dins l'`if`) la instrucció `cont++`.

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.

Solució: L'ús del vector `ind` fa que varies iteracions del bucle puguen acabar accedint a un mateix element del vector `x`. Per eixa raó, el càlcul del màxim requereix d'una secció crítica per a evitar condicions 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;
         }
      }
   }
}
```

L'avaluació de la funció `f2` es guarda en una variable auxiliar per a evitar ser realitzada varies vegades. A més, seria molt ineficient realitzar l'avaluació dins la secció crítica, donat que es faria de forma seqüencial.

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.

Solució: En cas de varies ocurrències, el codi proporcionat retorna l'última posició, la qual pot obtenir-se en el codi paral·lel mitjançant el màxim.

```
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 solució anterior no funciona en versions d'OpenMP anteriors a la 3.1. Una solució alternativa és:

```
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;
}
```

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.

Solució:

```
#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 paral·lelització la situem en el bucle més extern per a una major granularitat i menor temps de sincronització.

La paral·lelització genera una condició de carrera en l'actualització del màxim de les sumes per files. Per a evitar errors en les actualitzacions simultànies s'utilitza una secció crítica que evita la modificació concurrent de `norm`. Per a evitar una excessiva seqüencialització s'inclou la secció crítica dins de la comprovació del màxim, repetint-se la comprovació dins de la secció crítica per a assegurar que només es modifica el valor de `norm` si és un valor superior a l'acumulat.

Una solució alternativa seria modificar la directiva `parallel` com es mostra a continuació (amb la qual cosa no seria necessària la directiva `critical`). No obstant, aquesta variant no seria vàlida en versions antigues d'OpenMP (anteriors a 3.1), on no estava permesa la reducció amb l'operador `max`.

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

(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.

Solució: $t(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 2 = 2n^2$ 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 l'speedup i l'eficiència del codi paral·lel executat en p processadors.

Solució: $S(n,p) = \frac{t(n)}{t(n,p)} = p \qquad E(n,p) = \frac{S(n,p)}{p} = 1$
L'acceleració és la màxima esperable i l'eficiència és del 100%, la qual cosa indica un aprofitament màxim dels 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.

Solució:

```
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) Sense utilitzar reducció.

Solució: En aquest apartat presentem dues implementacions diferents, comentant quina és la més eficient.

```
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 implementació `prod3` és més eficient que la implementació `prod2`, perquè el nombre d'operacions `atomic` que fa és menor.

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`.

Solució:

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

Solució: La solució és realitzar una regió paral·lela en la que es reparteixen les iteracions de forma manual en base al nombre de fils i a l'identificador de fil. En aquest cas no ens està permés utilitzar la clàusula `reduction`; una possible alternativa seria protegir l'accés a la variable compartida `equal` amb una construcció `atomic`.

```
int cmp(int n, double x[], double y[], int z[])
{
  int i, v, equal=0, jo, nh;
  double aux;
  #pragma omp parallel private(aux,i,v,jo)
  {
    nh = omp_get_num_threads();  /* nombre de fils */
    jo = omp_get_thread_num();   /* identificador de fil */
    for (i=jo; 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;
}
```

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.

Solució: L'ús del vector `ind` fa que varies iteracions del bucle `i` puguen acabar actualitzant un mateix element del vector `x`. Per aquesta raó, l'actualització deu fer-se en exclusió mútua, mitjantçant l'ús 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) Realitza una implementació paral·lela mitjançant OpenMP, en la qual es reparteixen les iteracions del bucle intern.

Solució:

```
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) 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?

Solució: El bucle `j` fa `i` iteracions, amb la qual cosa les iteracions del bucle `i` seran més costoses com més gran siga el valor de `i`. Per tant, una planificació estàtica amb un únic *chunk* per fil no seria adequada. Seria millor una planificació cíclica (estàtica amb `chunk`=1), dinàmica o *guided*.

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.

Solució: El fragment de codi que calcula el màxim és independent del fragment que calcula el mínim, i per tant es podria pensar en una paral·lelització basada en `sections`, però es descarta aquesta opció perquè sols s'aprofitarien 2 fils, i l'enunciat demana una paral·lelització eficient per a qualsevol nombre de fils. Per tant, es fa una paral·lelització de cadascun dels bucles. Donat que el primer bucle és independent del segon, fem ús de la clàusula `nowait` per a evitar una sincronització entre els dos.

La solució utilitzant reduccions seria la següent.

```
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;
    }
  }
}
```

Com alternativa a les reduccions, es poden usar seccions crítiques, encara que seria menys eficient.

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

Solució:

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

        #pragma omp for nowait
        ...
```

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.

Solució:

```
int funcio(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 tercer dels bucles paral·lelitzats es calcula un màxim i la seua posició. Les variables compartides `pos_max` i `max` poden ser actualitzades simultàniament en diferents iteracions, per la qual cosa es requereix exclusió mútua en l'accés a aquestes (construcció `critical`). A més, la repetició de la instrucció

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

abans de la secció crítica té per objecte millorar l'eficiència, evitant entrar en aquesta secció en cas que no siga necessari.

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

Solució: No. En el primer no, perquè fins que no acaben tots els fils no es pot calcular la norma correctament. En el segon no, perquè el tercer bucle necessita els valors de `v[i]` actualitzats del segon. En el tercer tampoc, perquè no hi ha res paral·lel darrere.

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

Solució: Hi hauria que afegir en les directives `for` una clàusula de planificació apropiada, como pot ser `schedule(static,2)` o `schedule(dynamic,2)` que indique que es repartisquen les iteracions entre els fils de 2 en 2.

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.

Solució: La forma més senzilla d'implementar-ho és mitjançant una reducció (suposant OpenMP versió 3.1 o posterior). A més, cal sincronitzar l'accés concurrent a la variable `saldos`.

```
double transferencies(double saldos[], int origens[], int destins[],
       double quantitats[], int n)
{
   int i, i1, i2;
   double diners, maxtransf=0;
   #pragma omp parallel for private(i1,i2,diners) reduction(max:maxtransf)
   for (i=0; i<n; i++) {
```

```
            i1 = origens[i];
            i2 = destins[i];
            diners = quantitats[i];
            #pragma omp atomic
            saldos[i1] -= diners;
            #pragma omp atomic
            saldos[i2] += diners;
            if (diners>maxtransf) maxtransf = diners;
         }
         return maxtransf;
      }
```

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

Solució: En aquest cas no servix la directiva `reduction` i és necessari crear una secció crítica.

```
double transferencies(double saldos[], int origens[], int destins[],
       double quantitats[], int n)
{
   int i, i1, i2;
   double diners, maxtransf=0, imax;
   #pragma omp parallel for private(i1,i2,diners)
   for (i=0; i<n; i++) {
      i1 = origens[i];
      i2 = destins[i];
      diners = quantitats[i];
      #pragma omp atomic
      saldos[i1] -= diners;
      #pragma omp atomic
      saldos[i2] += diners;
      if (diners>maxtransf)
        #pragma omp critical
        if (diners>maxtransf) {
          maxtransf = diners;
          imax = i;
        }
   }
   printf("Transferència amb més diners=%d\n",imax);
   return maxtransf;
}
```

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.

Solució: El programa es pot dividir en tres tasques:

- Tasca 1: Modificació de la matriu `A` a partir de la pròpia matriu `A` (primer bucle niat).
- Tasca 2: Modificació de la matriu `B` a partir de la matriu `A` (segon bucle niat).
- Tasca 3: Obtenció de `maxi` $= \max_{\substack{0\leq i<N \\ 0\leq j<N}} \{b_{ij}^2\}$, sent b_{ij} l'element (i,j) de la matriu `B` (tercer bucle niat).

Com pot observar-se, la Tasca 2 depèn de la Tasca 1 i la Tasca 3 de la Tasca 2, per la qual cosa ha de finalitzar una tasca abans d'iniciar-se la següent. Per a paral·lelitzar el codi, paral·lelitzarem les tres tasques a nivell de bucles:

- Tasca 1: Com hi ha una dependència en les iteracions del bucle de variable iteradora `i` (els valors de la nova fila `i` de la matriu `A` depenen dels anteriors valors de la fila `i-1` de la matriu `A`) i sempre convé paral·lelitzar el més extern, intercanviem l'ordre dels bucles paral·lelitzant el més extern (paral·lelització per columnes).
- Tarea 2: Paral·lelitzem directament el bucle més extern (obtenció en paral·lel de les files de la matriu `B`).
- Tarea 3: Paral·lelitzem directament el bucle més extern. Per a obtindre el valor `maxi` es pot fer una reducció sobre `maxi` o usar regions crítiques.

A continuació es mostra la primera possibilitat:

```
double funciop(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;
    }
```

L'altra possibilitat consistiria en canviar el tercer bucle niat pel següent:

```
      #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 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.

Solució:

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

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

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

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

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.

Solució: En la paral·lelització del segon fragment de codi, hi ha que protegir l'accés a les variables `k` i `posicions` mitjançant una secció crítica, per a evitar condicions de carrera.

```
int funcio(double A[N][N],double posicions[][2])
{
  int i,j,k=0;
  double maxim;
  /* Calculem el màxim */
  maxim = A[0][0];
  #pragma omp parallel
  {
    #pragma omp for private(j) reduction(max:maxim)
    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 */
    #pragma omp for private(j)
    for (i=0;i<N;i++) {
      for (j=0;j<N;j++) {
        if (A[i][j] == maxim) {
          #pragma omp critical
          {
            posicions[k][0] = i;
            posicions[k][1] = j;
            k = k+1;
          }
        }
      }
    }
  }
  return k;
```

```
}
```

(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`.

Solució: Incorporariem les variables `id` i `num_maxims`.

```
int funcio(double A[][N],double posicions[][2])
{
  int i,j,k=0,id,num_maxims;
  double maxim;
  /* Calculem el màxim */
  maxim = A[0][0];
  #pragma omp parallel private(id,num_maxims)
  {
    #pragma omp for private(j) reduction(max:maxim)
    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 */
    id = omp_get_thread_num();
    num_maxims = 0;
    #pragma omp for private(j)
    for (i=0;i<N;i++) {
      for (j=0;j<N;j++) {
        if (A[i][j] == maxim) {
          #pragma omp critical
          {
            posicions[k][0] = i;
            posicions[k][1] = j;
            k = k+1;
          }
          num_maxims++;
        }
      }
    }
    printf("Fil %d: trobats %d valors màxims\n",id,num_maxims);
  }
  return k;
}
```

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];
  }
  ...
}
```

Solució: En el primer bucle, tenim que dues o més iteracions poden correspondre a actuacions d'un mateix jugador, i en eixe cas actualitzarien el mateix element dels vectors `suma_valoracio` i `num_partits`. Per a evitar problemes de condicions de carrera, l'actualització d'aquests vectors deurà fer-se per tant en exclusió mútua, mitjançant la clàusula `atomic`.

```
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];
  ...
  #pragma omp parallel
  {
    #pragma omp for private(jugador,punts,rebots,taps)
    for (i=0;i<NP*NJ;i++) {
      jugador = M[i][0];
      punts   = M[i][1];
      rebots  = M[i][2];
      taps    = M[i][3];
      #pragma omp atomic
      suma_valoracio[jugador] += punts+1.5*rebots+2*taps;
      #pragma omp atomic
      num_partits[jugador]++;
```

```
        if (punts>max_punts) {
          #pragma omp critical
          if (punts>max_punts) {
            max_punts = punts;
            max_anotador = jugador;
          }
        }
      }
      #pragma omp single
      printf("Maxim anotador: jugador %d (%d punts)\n",max_anotador,max_punts);
      #pragma omp for
      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);
}
```

Solució:

```
/* 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;
  #pragma omp parallel
  {
    #pragma omp for private(i,p)
    for (k = 0; k < nv; k++) {
      i = index[k]; p = punts[k];
      #pragma omp atomic
      totals[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("Les dues puntuacions més altes han sigut %d y %d.\n",max1,max2);
    #pragma omp for private(t) reduction(+:passen,total)
    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.

Solució:

```
void facturacions(int n, int usuaris[], int cansons[], float preus[],
                  float facturacio[], int ndescarregues[])
{
  int i,u,c,millor_canso=0;
  float p;
  #pragma omp parallel
  {
    #pragma omp for private(u,c,p)
    for (i=0;i<n;i++) {
       u = usuaris[i];
       c = cansons[i];
       p = preus[c];
       #pragma omp atomic
       facturacio[u] += p;
       #pragma omp atomic
       ndescarregues[c]++;
    }
    #pragma omp for
    for (i=0;i<NC;i++) {
       if (ndescarregues[i]>ndescarregues[millor_canso])
         #pragma omp critical
         if (ndescarregues[i]>ndescarregues[millor_canso])
           millor_canso = i;
    }
  }
  printf("La cançó %d és la més descarregada\n",millor_canso);
}
```

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

Solució: No seria vàlid emprar la directiva `nowait`, donat que el segon bucle només pot començar quan ja ha terminat el primer i hem completat el nombre de vegades que s'ha descarregat cada cançó.

(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.

Solució:

```
void facturacions(int n, int usuaris[], int cansons[], float preus[],
                  float facturacio[], int ndescarregues[])
{
```

```
      int i,u,c,millor_canso=0;
      float p;
      int myid,descarregues_fil;
      #pragma omp parallel private(myid,descarregues_fil)
      {
        myid = omp_get_thread_num();
        descarregues_fil = 0;
        #pragma omp for private(u,c,p)
        for (i=0;i<n;i++) {
           u = usuaris[i];
           c = cansons[i];
           p = preus[c];
           #pragma omp atomic
           facturacio[u] += p;
           #pragma omp atomic
           ndescarregues[c]++;
           descarregues_fil++;
        }
        printf("El fil %d ha gestionat %d descarregues\n",myid, descarregues_fil);
        #pragma omp for
        for (i=0;i<NC;i++) {
           if (ndescarregues[i]>ndescarregues[millor_canso])
             #pragma omp critical
             if (ndescarregues[i]>ndescarregues[millor_canso])
               millor_canso = i;
        }
      }
      printf("La cançó %d és la més descarregada\n",millor_canso);
    }
```

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.

Solució:

```
void histograma(int histo[], float notes[], int n) {
  int i, nota;
  float rnota;
  for (i=0;i<5;i++) histo[i] = 0;
  #pragma omp parallel for private (nota, rnota)
  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 */
    #pragma omp atomic
    histo[nota]++;
  }
}
```

(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).

Solució:

```
void histograma(int histo[], float notes[], int n) {
  int i, nota, imax;
  float vmin, vmax, rnota;

  for (i=0;i<5;i++) histo[i] = 0;

  vmin = notes[0];
  vmax = notes[0];
  imax = 0;
  #pragma omp parallel for private (nota, rnota) reduction (min:vmin)
  for (i=0;i<n;i++) {
    rnota = round(notes[i]*10)/10.0;
    if (notes[i]>vmax)
      #pragma omp critical
      if (notes[i]>vmax) {
        imax = i;
        vmax = notes[i];
      }

    if (notes[i]<vmin) vmin = notes[i];

    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 */

        #pragma omp atomic
        histo[nota]++;
      }

      printf("La millor nota es %f, obtinguda per %d, i la pitjor nota es %f\n",
             vmax, imax, vmin);
    }
```

Encara que és menys eficient, el càlcul del mínim també prodria fer-se amb un `critical`, de forma similar al màxim (açò seria necessari en versions antigues d'OpenMP que no permeten reducció de mínim). En eixe cas, seria convenient utilitzar `critical` amb nom, per exemple `critical (maxim)` i `critical (minim)`.

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.

Solució:

```
struct viatge {
   int estacio_origen;
   int estacio_desti;
   float temps_minuts;
};
void actualiza_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;
   #pragma omp parallel for private(origen,desti,temps) reduction(+:tmitja)
   for (i=0;i<num_viatges;i++) {
      origen  = viatges[i].estacio_origen;
      desti = viatges[i].estacio_desti;
      temps  = viatges[i].temps_minuts;
      #pragma omp atomic
      num_bicis[origen]--;
      #pragma omp atomic
      num_bicis[desti]++;
      tmitja += temps;
      if (temps>tmax) {
         #pragma omp critical (maxim)
         if (temps>tmax) {
            tmax=temps; ormax=origen; destmax=desti;
         }
      }
      if (temps<tmin) {
         #pragma omp critical (minim)
         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);
}
```