**Computació Paral · lela**
Grau en Enginyeria Informàtica (ETSInf)
Curs 2012-13 ◇ Examen parcial 5/11/2012 ◇ Duració: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

**Qüestió 1** (0.5 punts)

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

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

**Qüestió 2** (0.75 punts)

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;
}
```

0.4 p. (a) Realitza una implementació paral · lela mitjançant OpenMP d'aquest algorisme. Justifica la raó per la qual introdueixes cada canvi.

0.2 p. (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.

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

**Qüestió 3** (1.25 punts)
Donada la següent funció:

```
double funcio(int n, double u[], double v[], double w[], double z[])
{
  int i;
  double sv,sw,res;

  calcula_v(n,v);         /* tasca 1 */
  calcula_w(n,w);         /* tasca 2 */
  calcula_z(n,z);         /* tasca 3 */
  calcula_u(n,u,v,w,z);   /* tasca 4 */
  sv = 0;
  for (i=0; i<n; i++)  sv = sv + v[i];         /* tasca 5 */
  sw = 0;
  for (i=0; i<n; i++)  sw = sw + w[i];         /* tasca 6 */
  res = sv+sw;
  for (i=0; i<n; i++)  u[i] = res*u[i];        /* tasca 7 */
  return res;
}
```

Les funcions `calcula_X` tenen com a entrada els vectors que reben com a arguments i amb ells modifiquen el vector X indicat. Cada funció únicament modifica el vector que apareix en el seu nom. Per exemple, la funció `calcula_u` utilitza els vectors `v`, `w` i `z` per a realitzar uns càlculs que guarda en el vector `u`, però no modifica ni `v`, ni `w`, ni `z`.

Açò implica, per exemple, que les funcions `calcula_v`, `calcula_w` i `calcula_z` són independents i podrien realitzar-se simultàniament. No obstant açò, la funció `calcula_u` necessita que hagen acabat les altres, perquè usa els vectors que elles emplenen (`v`,`w`,`z`).

0.2 p. (a) Dibuixa el graf de dependències de les diferents tasques.

0.75 p. (b) Paral · lelitza la funció de forma eficient.

0.3 p. (c) Si suposem que el cost de totes les funcions `calcula_X` és el mateix i que el cost dels bucles posteriors és menyspreable, quin seria el speedup màxim possible?

**Computació Paral · lela**
Grau en Enginyeria Informàtica (ETSInf)
Curs 2012-13 ◇ Examen final 21/1/2013 ◇ Duració: 3h

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

**Qüestió 1** (1.25 punts)
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;
}
```

0.2 p. (a) Indica el seu cost teòric (en flops).

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

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

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

**Qüestió 2** (0.75 punts)
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;
}
```

0.4 p. (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?

0.15 p. (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 $n$.

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

**Qüestió 3** (0.5 punts)

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

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

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

**Qüestió 4** (1 punt)

Es vol paral · lelitzar el següent codi mitjançant MPI. Suposem que es disposa de 3 procesos.

```
double a[N],b[N],c[N],v=0.0,w=0.0;
T1(a,&v);
T2(b,&w);
T3(b,&v);
T4(c,&w);
T5(c,&v);
T6(a,&w);
```

Totes les funcions lligen i modifiquen ambdos arguments, també els vectors. Suposem que els vectors `a`, `b` i `c` estan emmagatzemats en $P_0$, $P_1$ i $P_2$, respectivament, i son massa grans per a poder ser enviats eficientement d'un procés a un altre.

0.25 p. (a) Dibuixa el graf de dependències de les diferents tasques, indicant quina tasca s'assigna a cada procés.

0.75 p. (b) Escriu el codi MPI que resol el problema.

**Qüestió 5** (1 punt)

La $\infty$-norma d'una matriu es defineix com el màxim de les sumes dels valors absoluts dels elements de cada fila: $\max_{i=1..n} \left\{ \sum_{j=0}^{m-1} |a_{i,j}| \right\}$. El següent codi seqüencial implementa aquesta operació per al cas d'una matriu quadrada.

```
#include <math.h>
#define N 800

double infNorm(double A[][N]) {
  int i,j;
  double s,nrm=0.0;

  for (i=0; i<N; i++) {
    s=0.0;
    for (j=0; j<N; j++)
      s+=fabs(A[i][j]);
    if (s>nrm)
      nrm=s;
  }
  return nrm;
}
```

0.5 p. (a) Implementar una versió paral · lela mitjançant MPI utilitzant operacions col · lectives en la mesura que siga possible. Assumir que la grandària del problema és múltiple exacte del nombre de processos. La matriu està inicialment emmagatzemada en $P_0$ i el resultat deu quedar també en $P_0$.

Nota: es suggerix utilitzar la següent capçalera per a la funció paral · lela, on `Alocal` és una matriu que se suposa ja reservada en memòria, i que pot ser utilitzada per la funció per a emmagatzemar la part local de la matriu `A`.

```
double infNormPar(double A[][N], double ALocal[][N])
```

0.25 p. (b) Obtindre el cost computacional i de comunicacions de l'algorisme paral · lel. Assumir que l'operació `fabs` té un cost menyspreable, així com les comparacions.

0.25 p. (c) Calcular el speed-up i l'eficiència quan la grandària del problema tendeix a infinit.

**Qüestió 6** (0.5 punts)

Siga `A` un array bidimensional de números reals de doble precisió, de dimensió `N`×`N`. Defineix un tipus de dades derivades MPI que permeta enviar una submatriu de dimensió $3 \times 3$. Per exemple, la submatriu que comença en `A[0][0]` serien els elements marcats amb $\star$:

$$\mathtt{A} = \begin{bmatrix} \star & \star & \star & \cdot & \cdot & \cdot \\ \star & \star & \star & \cdot & \cdot & \cdot \\ \star & \star & \star & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}.$$

(a) Realitza les corresponents crides per a l'enviament des de $P_0$ i la recepció en $P_1$ del bloc de la figura.

(b) Indica què caldria modificar en el codi anterior perquè el bloc enviat per $P_0$ siga el que comença en la posició (0,3), i que es reba en $P_1$ sobre el bloc que comença en la posició (3,0).

**Computació Paral · lela**
Grau en Enginyeria Informàtica (ETSInf)
Curs 2013-14 ◇ Examen parcial 11/11/2013 ◇ Duració: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

**Qüestió 1** (1 punt)

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;
}
```

0.4 p. (a) Paral · lelitza-la utilitzant construccions de tipus `parallel for`.

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

**Qüestió 2** (0.8 punts)

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];
    }
  }
}
```

0.4 p. (a) Paral · lelitza-la amb OpenMP usant dues regions paral · leles.

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

**Qüestió 3** (1.2 punts)

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 */
```

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

0.4 p. (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.

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

**Computació Paral · lela**
Grau en Enginyeria Informàtica (ETSInf)
Curs 2013-14 ◇ Examen parcial 13/1/2014 ◇ Duració: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

**Qüestió 1** (1.2 punts)

El següent programa compta el nombre d'ocurrències d'un valor en una matriu.

```
#include <stdio.h>
#define DIM 1000

void llegir(double A[DIM][DIM], double *x)
{ ... }

int main(int argc, char *argv[])
{
  double A[DIM][DIM], x;
  int i,j,cont;

  llegir(A,&x);
  cont=0;
  for (i=0; i<DIM; i++)
    for (j=0; j<DIM; j++)
      if (A[i][j]==x) cont++;
  printf("%d ocurrencies\n", cont);
  return 0;
}
```

0.8 p. (a) Fes una versió paral · lela MPI del programa anterior, utilitzant operacions de comunicació col · lectiva quan siga possible. La funció `llegir` haurà de ser invocada solament pel procés 0. Es pot assumir que `DIM` és divisible entre el nombre de processos. Nota: cal escriure el programa complet, incloent la declaració de les variables i les crides necessàries per a iniciar i tancar MPI.

0.4 p. (b) Calcula el temps d'execució paral · lel, suposant que el cost de comparar dos nombres reals és d'1 flop. Nota: per al cost de les comunicacions, suposar una implementació senzilla de les operacions col · lectives

**Qüestió 2** (1 punt)

En un programa MPI, un vector $x$, de dimensió $n$, es troba distribuït de forma cíclica entre $p$ processos, i cada procés guarda en l'array `xloc` els elements que li corresponen.

Implementa la següent funció, de manera que en ser invocada per tots els processos, realitze les comunicacions necessàries perquè el procés 0 arreplegue en l'array `x` una còpia del vector complet, amb els elements correctament ordenats segons l'índex global.

Cada procés de l'1 al $p-1$ haurà d'enviar al procés 0 tots els seus elements mitjançant un únic missatge.

```
void comunica_vec(double xloc[], int n, int p, int rank, double x[])
/* rank es l'index del proces local                        */
/* Aquesta funcio assumeix que n es multiple exacte de p */
```

**Qüestió 3** (0.8 punts)

El següent fragment de codi és incorrecte (des del punt de vista semàntic, no perquè hi haja un error en els arguments). Indica per què i proposa dues solucions diferents.

```
MPI_Status stat;
int sbuf[N], rbuf[N], rank, size, src, dst;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
src = (rank==0)? size-1: rank-1;
dst = (rank==size-1)? 0: rank+1;
MPI_Ssend(sbuf, N, MPI_INT, dst, 111, MPI_COMM_WORLD);
MPI_Recv(rbuf, N, MPI_INT, src, 111, MPI_COMM_WORLD, &stat);
```

**Computació Paral · lela**
Grau en Enginyeria Informàtica (ETSInf)
Curs 2013-14 ◇ Examen final 27/1/14 ◇ Bloc OpenMP ◇ Duració: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

**Qüestió 1** (1 punt)

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;
  }
}
```

0.75 p. (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.

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

**Qüestió 2** (1 punt)

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;
}
```

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

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

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

**Qüestió 3** (1 punt)

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;
}
```

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

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

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

**Qüestió 4** (0.8 punts)

Volem mesurar la latència d'un anell de `p` processos en MPI, entenent per latència el temps que tarda un missatge de grandària 0 a circular entre tots els processos. Un anell de `p` processos MPI funciona de la següent manera: $P_0$ envia el missatge a $P_1$, quan aquest el rep, el reenvía a $P_2$, i així successivament fins que arriba a $P_{p-1}$ que l'enviarà a $P_0$. Escriu un programa MPI que implemente aquest esquema de comunicació i mostre la latència. És recomanable fer que el missatge done més d'una volta a l'anell, i després traure el temps mitjà per volta, per a obtenir una mesura més fiable.

**Qüestió 5** (1.2 punts)

El següent programa paral · lel MPI deu calcular la suma de dues matrius $A$ i $B$ de dimensions $M \times N$ utilitzant una distribució cíclica de files, suposant que el nombre de processos `p` és divisor de $M$ i tenint en compte que $P_0$ té emmagatzemades inicialment les matrius $A$ i $B$.

```
int p, rank, i, j, mb;
double A[M][N], B[M][N], Al[M][N], Bl[M][N];

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank==0) llegir(A,B);

/* (a) Repartiment cíclic de files d'A i B */
/* (b) Càlcul local de Al+Bl */
/* (c) Recollida de resultats en el procés 0 */

if (rank==0) escriure(A);
MPI_Finalize();
```

0.5 p. (a) Implementa el repartiment cíclic de files de les matrius $A$ i $B$, sent `Al` i `Bl` les matrius locals. Per a realitzar aquesta distribució deus o bé definir un nou tipus de dades de MPI o bé usar comunicacions col · lectives.

0.2 p. (b) Implementa el càlcul local de la suma `Al+Bl`, emmagatzemant el resultat en `Al`.

0.5 p. (c) Escriu el codi necessari perquè $P_0$ emmagatzeme en `A` la matriu $A + B$. Per a açò, $P_0$ ha de rebre de la resta de processos les matrius locals `Al` obtingudes en l'apartat anterior.

**Qüestió 6** (1 punt)

Es vol implementar el càlcul de la $\infty$-norma d'una matriu quadrada, que s'obté com el màxim de les sumes dels valors absoluts dels elements de cada fila, $\max_{i=0}^{n-1} \left\{ \sum_{j=0}^{n-1} |a_{i,j}| \right\}$. Per a açò, es proposa un esquema mestre-treballadors. A continuació, es mostra la funció corresponent al

mestre (el procés amb identificador 0). La matriu s'emmagatzema per files en un array unidimensional, i suposem que és molt dispersa (té molts zeros), per la qual cosa el mestre envia únicament els elements no nuls (funció `comprimeix`).

```
int comprimeix(double *A,int n,int i,double *buf)
{
  int j,k = 0;
  for (j=0;j<n;j++)
    if (A[i*n+j]!=0.0) { buf[k] = A[i*n+j]; k++; }
  return k;
}
double mestre(double *A,int n)
{
  double buf[n];
  double norma=0.0,valor;
  int fila,complets=0,size,i,k;
  MPI_Status status;
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  for (fila=0;fila<size-1;fila++) {
    if (fila<n) {
      k = comprimeix(A, n, fila, buf);
      MPI_Send(buf, k, MPI_DOUBLE, fila+1, TAG_FILA, MPI_COMM_WORLD);
    } else
      MPI_Send(buf, 0, MPI_DOUBLE, fila+1, TAG_END, MPI_COMM_WORLD);
  }
  while (complets<n) {
    MPI_Recv(&valor, 1, MPI_DOUBLE, MPI_ANY_SOURCE, TAG_RESU,
             MPI_COMM_WORLD, &status);
    if (valor>norma) norma=valor;
    complets++;
    if (fila<n) {
      k = comprimeix(A, n, fila, buf);
      fila++;
      MPI_Send(buf, k, MPI_DOUBLE, status.MPI_SOURCE, TAG_FILA, MPI_COMM_WORLD);
    } else
      MPI_Send(buf, 0, MPI_DOUBLE, status.MPI_SOURCE, TAG_END, MPI_COMM_WORLD);
  }
  return norma;
}
```

Implementa la part dels processos treballadors, completant la següent funció:

```
void treballador(int n)
{
  double buf[n];
```

Nota: Per al valor absolut es pot usar

```
double fabs(double x)
```

Recorda que `MPI_Status` conté, entre altres, els camps `MPI_SOURCE` i `MPI_TAG`.

**Computació Paral · lela**
Grau en Enginyeria Informàtica (ETSInf)
Curs 2014-15 ◇ Examen parcial 3/11/2014 ◇ Duració: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

**Qüestió 1** (1 punt)

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;
}
```

0.6 p. (a) Paral · lelitza-la eficientment mitjançant OpenMP, utilitzant una sola regió paral · lela.

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

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

**Qüestió 2** (1 punt)

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 */
```

0.3 p. (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.

0.4 p. (b) Paral · lelitza el codi mitjançant OpenMP.

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

**Qüestió 3** (1 punt)

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;
}
```

0.6 p. (a) Paral · lelitza-la amb OpenMP, usant una única regió paral · lela.

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

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

**Computació Paral · lela**
Grau en Enginyeria Informàtica (ETSInf)
Curs 2014-15 ◇ Examen parcial 12/1/2015 ◇ Duració: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

**Qüestió 1** (1 punt)

La següent funció mostra per pantalla el màxim d'un vector `v` de `n` elements i la seua posició:

```
void func(double v[], int n) {
   double max = v[0];
   int i, posmax = 0;
   for (i=1; i<n; i++) {
      if (v[i]>max) {
         max = v[i];
         posmax=i;
      }
   }
   printf("Màxim: %f. Posició: %d\n", max, posmax);
}
```

Escriu una versió paral · lela MPI amb la següent capçalera, on els arguments `rank` i `np` han sigut obtinguts mitjançant `MPI_Comm_rank` i `MPI_Comm_size`, respectivament.

```
void func_par(double v[], int n, int rank, int np)
```

La funció ha d'assumir que l'array `v` del procés 0 contindrà inicialment el vector, mentre que en la resta de processos aquest array podrà usar-se per a emmagatzemar la part local que corresponga. Hauran de comunicar-se les dades necessàries de manera que el càlcul del màxim es repartisca de forma equitativa entre tots els processos. Finalment, només el procés 0 ha de mostrar el missatge per pantalla. S'han d'utilitzar operacions de comunicació punt a punt (no col · lectives).

Nota: es pot assumir que `n` és múltiple del nombre de processos.

**Qüestió 2** (1 punt)

0.6 p. (a) Implementa mitjançant comunicacions col · lectives una funció en MPI que sume dues matrius quadrades `a` i `b` i deixe el resultat en `a`, tenint en compte que les matrius `a` i `b` es troben emmagatzemades en la memòria del process $P_0$ i el resultat final també haurà d'estar en $P_0$. Suposarem que el nombre de files de les matrius (`N`, constant) és divisible entre el nombre de processos. La capçalera de la funció és:

```
void suma_mat(double a[N][N],double b[N][N])
```

0.4 p. (b) Determina el temps paral · lel, l'speed-up i l'eficiència de la implementació detallada en l'apartat anterior, indicant breument per al seu càlcul com es realitzarien cadascuna de les operacions col · lectives (nombre de missatges i grandària de cadascun d'ells). Pots assumir una implementació senzilla (no òptima) d'aquestes operacions de comunicació.

**Qüestió 3** (1 punt)

Implementa una funció on, donada una matriu $A$ de $N \times N$ nombres reals i un índex $k$ (entre 0 i $N-1$), la fila $k$ i la columna $k$ de la matriu es comuniquen des del procés 0 a la resta de processos (sense comunicar cap altre element de la matriu). La capçalera de la funció seria així:

```
void bcast_fila_col(double A[N][N], int k)
```

Hauràs de crear i usar un tipus de dades que represente una columna de la matriu. No és necessari que s'envien juntes la fila i la columna. Es poden enviar per separat.

**Computació Paral · lela**
Grau en Enginyeria Informàtica (ETSInf)
Curs 2014-15 ◇ Examen final 21/1/15 ◇ Bloc OpenMP ◇ Duració: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

**Qüestió 1** (1 punt)

Donada aquesta funció en C:

```
double fun( int n, double a[], double b[] )
{
  int i,ac,bc;
  double asuma,bsuma,cota;

  asuma = 0; bsuma = 0;
  for (i=0; i<n; i++)
    asuma += a[i];
  for (i=0; i<n; i++)
    bsuma += b[i];
  cota = (asuma + bsuma) / 2.0 / n;

  ac = 0; bc = 0;
  for (i=0; i<n; i++) {
    if (a[i]>cota) ac++;
    if (b[i]>cota) bc++;
  }
  return cota/(ac+bc);
}
```

0.7 p. (a) Paral · lelitza-la eficientment mitjantçant directives OpenMP (sense alterar el codi existent).

0.3 p. (b) Indica el cost a priori en flops, tant seqüencial com paral · lel (considerant només operacions en coma flotant i assumint que una comparació implica una resta). Quin seria l'speed-up i l'eficiència per a $p$ processadors? Assumiu que $n$ és un múltiple exacte de $p$.

**Qüestió 2** (1.3 punts)

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);
```

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

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

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

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

**Qüestió 3** (0.7 punts)

Paral · lelitza mitjançant OpenMP el següent fragment de codi, on `f` i `g` són dues funcions que prenen 3 arguments de tipus `double` i retornen un `double`, i `fabs` és la funció estàndard que retorna el valor absolut d'un `double`.

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

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

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

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

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

**Qüestió 4** (1 punt)

Donada la següent funció, on suposem que les funcions `T1`, `T3` i `T4` tenen un cost de $n$ i les funcions `T2` i `T5` de $2n$, on $n$ és un valor constant.

```
double exemple(int i,int j)
{
  double a,b,c,d,e;
  a = T1(i);
  b = T2(j);
  c = T3(a+b,i);
  d = T4(a/c);
  e = T5(b/c);
  return d+e;   /* T6 */
}
```

0.2 p. (a) Dibuixa el graf de dependències i calcula el cost seqüencial.

0.6 p. (b) Paral · lelitza-la usant MPI amb dos processos. Tots dos processos invoquen la funció amb el mateix valor dels arguments `i`, `j` (no és necessari comunicar-los). El valor de retorn de la funció ha de ser correcte en el procés 0 (no és necessari que estiga en tots dos processos).

0.2 p. (c) Calcula el temps d'execució en paral · lel (càlcul i comunicacions) i l'speedup amb dos processos.

**Qüestió 5** (1 punt)

Donat el següent fragment de codi d'un programa paral · lel:

```
int j, proc;
double A[NFIL][NCOL], col[NFIL];

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
```

Escriu el codi necessari per a definir un tipus de dades MPI que permeta que una columna qualsevol de `A` es puga enviar o rebre amb només un missatge. Utilitza el tipus de dades per a fer que el procés 0 li envie la columna `j` al procés `proc`, qui la rebrà sobre l'array `col`, li canviarà el signe als elements, i li la retornarà al procés 0, que la rebrà de nou sobre la columna `j` de la matriu `A`.

**Qüestió 6** (0.4 punts)

El següent fragment de codi utilitza primitives de comunicació punt a punt per a un patró de comunicació que pot efectuar-se mitjançant una única operació col · lectiva.

```
#define TAG 999
int i, k, sz, rank;
float z[LNZ], zfull[LNFULL];     /* suposem LNFULL>=LNZ*sz */
...
MPI_Comm_size(comm, &sz);
MPI_Comm_rank(comm, &rank);
if (!rank) {
  for (i=0;i<LNZ;i++) zfull[i] = z[i];
  for (k=1;k<sz;k++) {
    MPI_Recv(&zfull[k*LNZ], LNZ, MPI_FLOAT, k, TAG, comm,
             MPI_STATUS_IGNORE);
  }
} else {
  MPI_Send(z, LNZ, MPI_FLOAT, 0, TAG, comm);
}
```

Escriu la crida a la primitiva MPI de l'operació de comunicació col · lectiva equivalent.

**Qüestió 7** (0.6 punts)

Donada la següent crida a una primitiva de comunicació col · lectiva:

```
long int factor=..., producte;
MPI_Allreduce(&factor, &producte, 1, MPI_LONG, MPI_PROD, comm);
```

Escriu un fragment de codi equivalent (ha de realitzar la mateixa comunicació i operacions aritmètiques associades) però utilitzant únicament primitives de comunicació punt a punt.

**Computació Paral · lela**
Grau en Enginyeria Informàtica (ETSInf)
Curs 2015-16 ◇ Examen parcial 9/11/2015 ◇ Duració: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

**Qüestió 1** (1.2 punts)

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;
}
```

0.7 p. (a) Paral · lelitza el codi mitjançant OpenMP. Per què ho fas així? Es valoraran més aquelles solucions que siguen més eficients.

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

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

**Qüestió 2** (1 punt)

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.

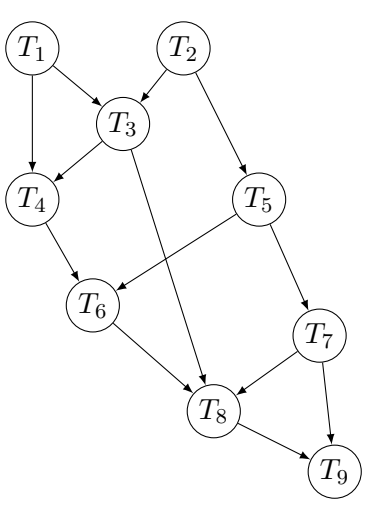
```
double transferencies(double saldos[], int origens[], int destins[],
       double quantitats[], int n)
{
   int i, i1, i2;
   double diners, maxtransf=0;

   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;
}
```

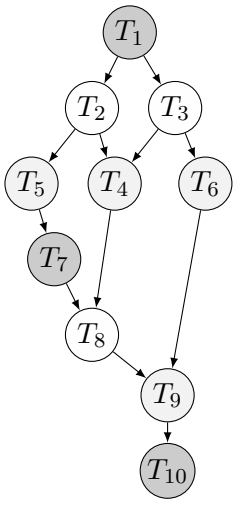
0.7 p. (a) Paral · leliza la funció de forma eficient mitjançant OpenMP.

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

**Qüestió 3** (0.8 punts)

Donats els següents grafs de dependències de tasques:




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

0.4 p. (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.

0.4 p. (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.

**Computació Paral · Iela**
Grau en Enginyeria Informàtica (ETSInf)
Curs 2015-16 ◇ Examen parcial 15/1/2016 ◇ Duració: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

**Qüestió 1** (1 punt)

Volem implementar una funció per a distribuir una matriu quadrada entre els processos d'un programa MPI, amb la següent capçalera:

```
void comunica(double A[N][N], double Aloc[][N], int proc_fila[N], int root)
```

La matriu `A` es troba inicialment en el procés `root`, i ha de distribuir-se per files entre els processos, de manera que cada fila `i` ha d'anar al procés `proc_fila[i]`. El contingut del array `proc_fila` és vàlid en tots els processos. Cada procés (inclós el `root`) ha d'emmagatzemar les files que li corresponguen en la matriu local `Aloc`, ocupant les primeres files (o siga, si a un procés se li assignen `k` files, aquestes han de quedar emmagatzemades en les primeres `k` files de `Aloc`).

Exemple per a 3 processos:

`A`

| | | | | |
|---|---|---|---|---|
| 11 | 12 | 13 | 14 | 15 |
| 21 | 22 | 23 | 24 | 25 |
| 31 | 32 | 33 | 34 | 35 |
| 41 | 42 | 43 | 44 | 45 |
| 51 | 52 | 53 | 54 | 55 |

`proc_fila`

| |
|---|
| 0 |
| 2 |
| 0 |
| 1 |
| 1 |

`Aloc` en $P_0$

| | | | | |
|---|---|---|---|---|
| 11 | 12 | 13 | 14 | 15 |
| 31 | 32 | 33 | 34 | 35 |

`Aloc` en $P_1$

| | | | | |
|---|---|---|---|---|
| 41 | 42 | 43 | 44 | 45 |
| 51 | 52 | 53 | 54 | 55 |

`Aloc` en $P_2$

| | | | | |
|---|---|---|---|---|
| 21 | 22 | 23 | 24 | 25 |

0.8 p. (a) Escriu el codi de la funció.

0.2 p. (b) En un cas general, es podria usar el tipus de dades *vector* de MPI (`MPI_Type_vector`) per a enviar a un procés totes les files que li toquen mitjançant un sol missatge? Si es pot, escriu les instruccions per a definir-lo. Si no es pot, justifica per què.

**Qüestió 2** (1 punt)

La següent funció calcula el producte escalar de dos vectors:

```
double scalarprod(double X[], double Y[], int n) {
  double prod=0.0;
  int i;
  for (i=0;i<n;i++)
    prod += X[i]*Y[i];
  return prod;
}
```

0.5 p. (a) Implementa una funció per a realitzar el producte escalar en paral · lel mitjançant MPI, utilitzant en la mesura del possible operacions col · lectives. Se suposa que les dades estan disponibles en el procés $P_0$ i que el resultat ha de quedar també en $P_0$ (el valor de tornada de la funció solament és necessari que siga correcte en $P_0$). Es pot assumir que la grandària del problema `n` és exactament divisible entre el nombre de processos.

Nota: a continuació es mostra la capçalera de la funció a implementar, incloent la declaració dels vectors locals (suposem que `MAX` és suficientment gran per a qualsevol valor de $n$ i nombre de processos).

```
double pscalarprod(double X[], double Y[], int n)
{
  double Xlcl[MAX], Ylcl[MAX];
```

0.3 p. (b) Calcula l'speed-up. Si per a una grandària suficientment gran de missatge, el temps d'enviament per element fóra equivalent a 0.1 flops, quin speed-up màxim es podria aconseguir quan la grandària del problema tendeix a infinit i per a un valor suficientment gran de processos?

0.2 p. (c) Modifica el codi anterior per a que el valor de retorn siga el correcte en tots els processos.

**Qüestió 3** (1 punt)

Es vol distribuir entre 4 processos una matriu quadrada d'ordre $2N$ ($2N$ files per $2N$ columnes) definida a blocs com

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix},$$

on cada bloc $A_{ij}$ correspon a una matriu quadrada d'ordre $N$, de manera que es vol que el procés $P_0$ emmagatzeme localment la matriu $A_{00}$, $P_1$ la matriu $A_{01}$, $P_2$ la matriu $A_{10}$ i $P_3$ la matriu $A_{11}$.

Per exemple, la següent matriu amb $N = 2$ quedaria distribuïda com es mostra:

$$A = \left(\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array}\right)$$

En $P_0$: $\begin{pmatrix} 1 & 2 \\ 5 & 6 \end{pmatrix}$ En $P_1$: $\begin{pmatrix} 3 & 4 \\ 7 & 8 \end{pmatrix}$

En $P_2$: $\begin{pmatrix} 9 & 10 \\ 13 & 14 \end{pmatrix}$ En $P_3$: $\begin{pmatrix} 11 & 12 \\ 15 & 16 \end{pmatrix}$

0.8 p. (a) Implementa una funció que realitze la distribució esmentada, definint per a açò el tipus de dades MPI necessari. La capçalera de la funció seria:

```
void comunica(double A[2*N][2*N], double B[N][N])
```

on `A` és la matriu inicial, emmagatzemada en el procés 0, i `B` és la matriu local on cada procés ha de emmagatzemar el bloc que li corresponga de $A$.

Nota: es pot assumir que el nombre de processos del comunicador és 4.

0.2 p. (b) Calcula el temps de comunicacions.

**Qüestió 1** (0.8 punts)

Donada la següent funció:

```
double fn1(double A[M][N], double b[N], double c[M], double z[N])
{
   double s, s2=0.0, elem;
   int i, j;

   for (i=0; i<M; i++) {
      s = 0.0;
      for (j=0; j<N; j++)
         s = s + A[i][j]*b[j];
      elem = s*s;
      if (elem>c[i])
         c[i] = elem;
      s2 = s2+s;
   }

   for (i=0; i<N; i++)
      z[i] = 2.0/3.0*b[i];

   return s2;
}
```

0.4 p. (a) Paral · lelitza-la amb OpenMP de forma eficient. Utilitza si és possible una sola regió paral · lela.

0.2 p. (b) Fes que cada fil mostre una línia amb el seu identificador i el nombre d'iteracions del primer bucle `i` que ha realitzat.

0.2 p. (c) En el primer bucle paral · lelitzat, justifica si cal esperar diferències en prestacions entre les següents planificacions per al bucle: `schedule(static)`, `schedule(static,1)`, `schedule(dynamic)`.

**Qüestió 2** (1.2 punts)

Donada la següent funció:

```
double f(int n, double vec[])
{
  double res, v[NMAX], w[NMAX];
  A(n,v,vec);          /* Copia vec en v, cost n */
  B(n,w,vec);          /* Copia vec en w, cost n */
```

```
  C(n,vec);            /* Actualitza vec, cost n */
  D(n,vec);            /* Actualitza vec, cost n */
  E(n,v);              /* Actualitza v, cost 3n  */
  F(n,w);              /* Actualitza w, cost 2n  */
  res = G(n,vec,v,w);  /* Calcula res, cost 3n   */
  return res;
}
```

0.5 p. (a) Dibuixa el graf de dependències, indicant el grau màxim de concurrència, la longitud del camí crític i el grau mitjà de concurrència.

0.5 p. (b) Paral · lelitza-la amb OpenMP.

0.2 p. (c) Calcula l'speedup i l'eficiència màxims si s'executa amb 2 fils.

**Qüestió 3** (1 punt)

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;
}
```

0.8 p. (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.

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

**Qüestió 4** (1 punt)

Siga el codi seqüencial:

```
int i, j;
double A[N][N];
for (i=0;i<N;i++)
    for(j=0;j<N;j++)
        A[i][j]= A[i][j]*A[i][j];
```

0.8 p. (a) Implementa una versió paral · lela equivalent utilitzant MPI, tenint en compte els següents aspectes:

- El procés $P_0$ obté inicialment la matriu `A`, realitzant la crida `llegir(A)`, sent `llegir` una funció ja implementada.
- La matriu `A` s'ha de distribuir per blocs de files entre tots els processos.
- Finalment $P_0$ ha de contenir el resultat en la matriu `A`.
- Utilitza comunicacions col · lectives sempre que siga possible.

Se suposa que $N$ és divisible entre el nombre de processos i que la declaració de les matrius usades és

```
double A[N][N], B[N][N];  /* B: matriu distribuida */
```

0.2 p. (b) Calcula l'speedup i l'eficiència.

**Qüestió 5** (0.8 punts)

Desenvolupa una funció que servisca per a enviar una submatriu des del procés 0 al procés 1, on quedarà emmagatzemada en forma de vector. S'ha d'utilitzar un nou tipus de dades, de manera que s'utilitze un únic missatge. Recordeu que les matrius en C estan emmagatzemades en memòria per files.

La capçalera de la funció serà així:

```
void envia(int m,int n,double A[M][N],double v[MAX],MPI_Comm comm)
```

Nota: s'assumeix que `m*n<==MAX` i que la submatriu a enviar comença en l'element `A[0][0]`.

Exemple amb $M = 4$, $N = 5$, $m = 3$, $n = 2$:

| `A` (en $P_0$) | | | | | | `v` (en $P_1$) |
|---|---|---|---|---|---|---|
| 1 | 2 | 0 | 0 | 0 | | |
| 3 | 4 | 0 | 0 | 0 | → | 1 2 3 4 5 6 |
| 5 | 6 | 0 | 0 | 0 | | |
| 0 | 0 | 0 | 0 | 0 | | |

**Qüestió 6** (1.2 punts)

Donat el següent programa seqüencial:

```
int calcula_valor(int num);  /* funció donada, definida en un altre lloc */

int main(int argc,char *argv[])
{
  int n,i,val,min;

  printf("Nombre d'iteracions: ");
  scanf("%d",&n);
  min = 100000;
  for (i = 1; i <= n; i++) {
    val = calcula_valor(i);
    if (val < min) min = val;
  }
  printf("Mínim: %d\n",min);
  return 0;
}
```

0.8 p. (a) Paral · lelitza'l mitjançant MPI usant operacions de comunicació punt a punt. L'entrada i eixida de dades ha de fer-la únicament el procés 0. Pot assumir-se que $n$ és un múltiple exacte del nombre de processos.

0.4 p. (b) Què es canviaria per a usar operacions de comunicació col · lectives allà on siga possible?

**Computació Paral · lela**
Grau en Enginyeria Informàtica (ETSInf)
Curs 2016-17 ◇ Examen parcial 9/11/2016 ◇ Duració: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

**Qüestió 1** (1 punt)
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 ($n = \texttt{N}$), mentres que el cost de cadascuna de les altres funcions és $n^2$ flops.

0.3 p. (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.

0.4 p. (b) Paral · lelitza la funció amb OpenMP.

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

**Qüestió 2** (0.9 punts)
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];
  }
}
```

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

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

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

**Qüestió 3** (1.1 punts)

La següent funció calcula els punts obtinguts pels distints equips d'una lliga de futbol.

Durant la temporada hi ha hagut `NJ` jornades, en cadascuna de les quals s'han jugat `NPJ` partits. La informació de tots els partits es troba emmagatzemada en la matriu `partits`, cada element de la qual és una estructura de dades (`SPartit`) que conté les dades del partit (equips que el juguen i gols de cadascun).

Amb eixa informació, la funció calcula els `punts` que li corresponen a cadascun dels `NE` equips, tenint en compte que un equip aconsegueix 3 punts si guanya un partit, 1 punt si l'empata, i zero si el perd.

A més, la funció torna el número de gols de la temporada i obté també, per a cada jornada, el màxim nombre de gols en un partit (array `maxg_jornada`).

```
int func(SPartit partits[NJ][NPJ], int punts[NE], int maxg_jornada[NJ]) {
   int i, j, maxg, eq1, eq2, g1, g2, gols_temporada;
   gols_temporada = 0;
   for (i=0; i<NJ; i++) {      /* per a cada jornada */
      maxg = 0;
      for (j=0; j<NPJ; j++) {  /* per a cada partit de la jornada */
         eq1 = partits[i][j].eq1;
         eq2 = partits[i][j].eq2;
         g1 = partits[i][j].gols1;
         g2 = partits[i][j].gols2;
         if (g1>g2)
            punts[eq1] += 3;  /* Equip 1 ha guanyat */
         else if (g1<g2)
            punts[eq2] += 3;  /* Equip 2 ha guanyat */
         else {
            punts[eq1] += 1;  /* Empat */
            punts[eq2] += 1;
         }
         if (g1+g2>maxg) maxg = g1+g2;
         gols_temporada += g1+g2;
      }
      maxg_jornada[i] = maxg;
   }
   return gols_temporada;
}
```

0.6 p. (a) Paral · lelitza (de forma eficient) el bucle i.

0.5 p. (b) Paral · lelitza (de forma eficient) el bucle `j`. Cal tindre en compte que en una mateixa jornada cada equip juga només un partit. Indica clarament els canvis respecte al codi original (no respecte a l'apartat anterior).

**Qüestió 1** (1 punt)

Es pretén distribuir amb MPI els blocs quadrats de la diagonal d'una matriu quadrada de dimensió $3 \cdot \mathrm{DIM}$ entre 3 processos. Si la matriu fora de dimensió 6 (DIM=2), la distribució seria com s'exemplifica:

$$\begin{pmatrix} a_{00} & a_{01} & \dots & \dots & \dots & \dots \\ a_{10} & a_{11} & \dots & \dots & \dots & \dots \\ \dots & \dots & a_{22} & a_{23} & \dots & \dots \\ \dots & \dots & a_{32} & a_{33} & \dots & \dots \\ \dots & \dots & \dots & \dots & a_{44} & a_{45} \\ \dots & \dots & \dots & \dots & a_{54} & a_{55} \end{pmatrix} \rightarrow \begin{array}{c} P_0 \\ \\ P_1 \\ \\ P_2 \end{array} \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} a_{44} & a_{45} \\ a_{54} & a_{55} \end{bmatrix}$$

Realitza una funció que permeta enviar els blocs amb el mínim nombre de missatges. Es proporciona la definició de la capçalera de la funció per a facilitar la implementació. El procés 0 té en `A` la matriu completa i després de la crida a la funció s'ha de tenir en `Alcl` de cada procés el bloc que li correspon. Utilitza primitives de comunicació punt a punt.

```
void SendBAD(double A[3*DIM][3*DIM], double Alcl[DIM][DIM]) {
```

**Qüestió 2** (1 punt)

El següent programa llig una matriu quadrada $A$ d'ordre $N$ i construeix, a partir d'ella, un vector $v$ de dimensió $N$ de manera que la seu component $i$-ésima, $0 \leq i < N$, és igual a la suma dels elements de la fila $i$-ésima de la matriu $A$. Finalment, el programa imprimeix el vector $v$.

```
int main(int argc, char *argv[])
{
  int i,j;
  double A[N][N],v[N];
  read_mat(A);
  for (i=0;i<N;i++) {
    v[i] = 0.0;
    for (j=0;j<N;j++)
      v[i] += A[i][j];
  }
  write_vec(v);
  return 0;
}
```

0.75 p. (a) Utilitza comunicacions col · lectives per a implementar un programa MPI que realitze el mateix càlcul, d'acord amb els següents passos:

- El procés $P_0$ llig la matriu $A$.
- $P_0$ reparteix la matriu $A$ entre tots els processos.
- Cada procés calcula la part local de $v$.
- $P_0$ arreplega el vector $v$ a partir de les parts locals de tots els processos.
- $P_0$ escriu el vector $v$.

Nota: Per a simplificar, es pot assumir que $N$ és divisible entre el nombre de processos.

0.25 p. (b) Calcula els temps seqüencial i paral · lel, sense tenir en compte les funcions de lectura i escriptura. Indica el cost que has considerat per a cadascuna de les operacions col · lectives realitzades.

**Qüestió 3** (1 punt)

Donada la següent funció, on suposem que les funcions `T1`, `T2` i `T3` tenen un cost de $7n$ i les funcions `T5` i `T6` de $n$, sent $n$ un valor constant.

```
double exemple(int val[3])
{
  double a,b,c,d,e,f;
  a = T1(val[0]);
  b = T2(val[1]);
  c = T3(val[2]);
  d = a+b+c;       /* T4 */
  e = T5(val[2],d);
  f = T6(val[0],val[1],e);
  return f;
}
```

0.25 p. (a) Dibuixa el graf de dependències i calcula el cost seqüencial.

0.5 p. (b) Paral · lelitza-la mitjançant MPI, suposant que hi ha tres processos. Tots els processos invoquen la funció amb el mateix valor de l'argument `val` (no és necessari comunicar-lo). El valor de retorn de la funció ha de ser correcte en el procés 0 (no és necessari que ho siga en els altres processos).

Nota: per a les comunicacions han d'utilitzar-se únicament operacions de comunicació col · lectiva.

0.25 p. (c) Calcula el temps d'execució paral · lel (càlcul i comunicacions) i l'speedup amb tres processos. Calcula també l'speedup asimptòtic, és a dir, el límit quan $n$ tendeix a infinit.

**Qüestió 1** (0.9 punts)

```
double func(double x[], double A[N][N])
{
  int i, j, hit;
  double elem, val=0;
  for (i=0; i<N; i++) {
    elem = x[i];
    hit = 0;
    j = N-1;
    while (j>=0 && !hit) {
      if (elem<sqrt(A[i][j])) hit = 1;
      j--;
    }
    if (hit) {
      elem *= 2.0;
      x[i] /= 2.0;
      val += elem;
    }
  }
  return val;
}
```

0.4 p. (a) Paral · lelitza amb OpenMP (de forma eficient) el bucle `i` de la funció donada.

0.5 p. (b) Paral · lelitza amb OpenMP (de forma eficient) el bucle `j` de la funció donada.

**Qüestió 2** (1.1 punts)

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;
}
```

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

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

**Qüestió 3** (1 punt)

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];
}
```

0.5 p. (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).

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

**Qüestió 4** (0.9 punts)

Donada una matriu de `NF` files i `NC` columnes, inicialment emmagatzemada en el procés 0, es vol fer un repartiment de la mateixa per blocs de columnes entre els processos 0 i 1, de manera que les primeres NC/2 columnes es queden en el procés 0 i la resta vagen al procés 1 (suposarem que `NC` és parell).

Implementa una funció amb la següent capçalera, que faça el repartiment indicat mitjançant MPI, definint el tipus de dades necessari perquè els elements que li toquen al procés 1 es transmeten mitjançant un sol missatge. Quan la funció acabe, tant el procés 0 com l'1 hauran de tindre en `Aloc` el bloc de columnes que els toca. És possible que el nombre de processos siga superior a 2. En eixe cas sols el 0 i l'1 hauran de tindre el seu bloc de matriu en `Aloc`.

```
void distribueix(double A[NF][NC], double Aloc[NF][NC/2])
```

**Qüestió 5** (1.1 punts)

Donada la següent funció seqüencial:

```
int comptar(double v[], int n)
{
  int i, cont=0;
  double mitja=0;

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

  for (i=0;i<n;i++)
    if (v[i]>mitja/2.0 && v[i]<mitja*2.0)
      cont++;

  return cont;
}
```

0.7 p. (a) Fes una versió paral · lela utilitzant MPI, suposant que el vector `v` es troba inicialment només en el procés 0, i el resultat retornat per la funció només fa falta que siga correcte en el procés 0. Hauran de distribuir-se les dades necessàries perquè tots els càlculs es repartisquen de forma equitativa. Nota: Es pot assumir que `n` és divisible entre el nombre de processos.

0.4 p. (b) Calcula el temps d'execució de la versió paral · lela de l'apartat anterior, així com el límit del speedup quan `n` tendeix a infinit. Si has fet servir operacions col · lectives, indica quin és el cost que has considerat per a cadascuna d'elles.

**Qüestió 6** (1 punt)

Desenvolupa un programa *ping-pong*.

Es desitja un programa paral · lel, per a ser executat en 2 processos, que repetisca 200 vegades l'enviament del procés 0 al procés 1 i la devolució del procés 1 al 0, d'un missatge de 100 enters. Al final s'haurà de mostrar per pantalla el temps mitjà d'enviament d'un enter, calculat a partir del temps d'enviar/rebre tots eixos missatges.

El programa pot començar així:

```
int main(int argc,char *argv[])
{
  int v[100];
```

0.8 p. (a) Implementa el programa indicat.

0.2 p. (b) Calcula el temps de comunicacions (teòric) del programa.

**Computació Paral · lela**
Grau en Enginyeria Informàtica (ETSInf)
Curs 2017-18 ◇ Examen parcial 8/11/2017 ◇ Duració: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

**Qüestió 1** (1 punt)

La següent funció calcula una matriu $A$ a partir d'un array de nodes (array `node`). Per a cada node es calcula un coeficient, amb el qual es modifiquen tres elements de A, dos en la diagonal i un altre fora de la diagonal.

```
double funcio(double A[DIM][DIM], NodeData node[], int n)
{
  int i,f,c;
  double coef, sum;
  preparar(A);
  sum = 0;
  for (i=0; i<n; i++) {
    f = node[i].fila;
    c = node[i].columna;
    coef = calculacoef(node[i]);
    /* Modificar elements en la diagonal de A */
    A[f][f] += coef;
    A[c][c] += coef;
    /* Modificar element fora de la diagonal de A */
    A[f][c] -= coef;
    sum += coef;
  }
  return sum;
}
```

0.8 p. (a) Paral · lelitza la funció de forma eficient amb OpenMP. Se sap que dos nodes qualsevol poden modificar un mateix element de la diagonal, però no un mateix element fora de la diagonal. Justifica breument per què ho fas així.

0.2 p. (b) Calcula el temps d'execució a priori seqüencial, suposant que la funció `preparar` té un cost de $n$ flops i `calculacoef` té un cost de 5 flops. Calcula el temps d'execució paral · lel i l'speed-up. Calcula l'speed-up màxim si es disposara d'infinits processadors.

**Qüestió 2** (0.8 punts)

Donat el següent codi:

```
int funcio(double A[M][N])
{
  int i,j,resposta=1;
  double sumf,diag,elem;
  i = 0;
  while (i<M && resposta==1) {
```

```
    sumf = 0;
    diag = fabs(A[i][i]);
    for (j=0;j<N;j++) {
      if (i!=j) {
        elem = fabs(A[i][j]);
        sumf += elem;
      }
    }
    if (diag<sumf) resposta = 0;
    i++;
  }
  return resposta;
}
```

0.5 p. (a) Indica com es paral · lelitzaria mitjançant OpenMP, de la forma més eficient, el bucle exterior (`while`).

0.3 p. (b) Paral · lelitza el segon bucle (`for`).

**Qüestió 3** (1.2 punts)

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;
}
```

0.3 p. (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.

0.4 p. (b) Paral · lelitza-la amb OpenMP.

0.3 p. (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.

0.2 p. (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.

**Computació Paral · lela**
Grau en Enginyeria Informàtica (ETSInf)
Curs 2017-18 ◇ Examen parcial 15/1/2018 ◇ Duració: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

**Qüestió 1** (1 punt)

Donada la següent funció:

```
double calculs(double v[n])
{
  double x,y,a,b,c;
  x = f1(v);        /* tasca 1, cost n     */
  y = f2(v);        /* tasca 2, cost n     */
  a = f3(v,x+y);    /* tasca 3, cost 2n+1  */
  b = f4(v,x-y);    /* tasca 4, cost 3n+1  */
  c = a + b;        /* tasca 5              */
  return c;
}
```

0.1 p. (a) Sabent que cap de les funcions cridades modifica el vector `v`, dibuixa el graf de dependències.

0.7 p. (b) Paral · lelitza-la amb MPI utilitzant únicament operacions de comunicació punt a punt. El vector `v` ja està replicat en tots els processos. El resultat es necessita només en el procés 0. Assegura't de paral · lelitzar-lo de forma que no es puguen produir interbloquejos.

0.2 p. (c) Indica el temps teòric de l'algoritme de l'apartat anterior (incloent també les comunicacions).

**Qüestió 2** (1.1 punts)

El següent codi proporciona el resultat de l'operació $C = aA + bB$, on $A, B$ i $C$ són matrius de $M \times N$ components, i $a$ i $b$ són nombres reals:

```
int main(int argc, char *argv[]) {
  int i, j;
  double a, b, A[M][N], B[M][N], C[M][N];
  LligOperands(A, B, &a, &b);
  for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
      C[i][j] = a*A[i][j] + b*B[i][j];
    }
  }
  EscriuMatriu(C);
  return 0;
}
```

Desenvolupa una versió paral · lela mitjançant MPI utilitzant operacions col · lectives, tenint en compte que:

- $P_0$ obtindrà inicialment les matrius $A$ i $B$, així com els nombres reals $a$ i $b$, mitjançant la invocació de la funció `LligOperands`.
- Únicament $P_0$ haurà de disposar de la matriu $C$ completa com a resultat, i serà l'encarregat de cridar la funció `EscriuMatriu`.
- $M$ és un múltiple exacte del nombre de processos.
- Les matrius $A$ i $B$ s'hauran de distribuir cíclicament per files entre els processos, per tal de fer en paral · lel l'operació esmentada.

**Qüestió 3** (0.9 punts)

Es vol implementar una operació de comunicació entre 3 processos MPI en la qual el procés $P_0$ té emmagatzemada una matriu `A` de dimensió $N \times N$, i ha d'enviar al procés $P_1$ la submatriu formada per les files d'índex parell, i al procés $P_2$ la submatriu formada per les files d'índex imparell. S'hauran d'utilitzar tipus de dades derivats de MPI per tal de realitzar el menor nombre possible d'enviaments. Cada matriu recollida en $P_1$ i $P_2$ haurà de quedar emmagatzemada en la matriu local `B` de dimensió $N/2 \times N$. Nota: Es pot assumir que $N$ és un nombre parell.

Exemple: Si la matriu emmagatzemada en $P_0$ és

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$$

la matriu recollida per $P_1$ haurà de ser

$$B = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

i la matriu recollida per $P_2$ haurà de ser

$$B = \begin{pmatrix} 5 & 6 & 7 & 8 \\ 13 & 14 & 15 & 16 \end{pmatrix}.$$

0.7 p. (a) Implementa una funció amb la següent capçalera per a fer l'operació descrita:

```
void comunica(double A[N][N], double B[N/2][N])
```

0.2 p. (b) Calcula el temps de comunicacions.

**Qüestió 1** (1.1 punts)

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)    */
}
```

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

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

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

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

**Qüestió 2** (0.7 punts)

Siga el següent codi:

```
int i,j,k;
double a,b,aux,A[N][N],B[N][N],C[N][N];
scanf("%lf",&a);
scanf("%lf",&b);
genera1(A,a);    /* Cost igual a N^3   */
genera2(B,b);    /* Cost igual a 2*N^3 */
for (i=0; i<N; i++) {
   for (j=0; j<N; j++) {
      aux = 0.0;
      for (k=0; k<N; k++)
         aux += A[i][k]*B[k][j];
      C[i][j] = aux;
   }
}
```

on `genera1` i `genera2` són funcions prèviament declarades i definides, de cost $N^3$ i $2N^3$, respectivament. Paral · lelitza el codi mitjançant OpenMP de la forma més eficient possible, utilitzant una sola regió paral · lela.

**Qüestió 3** (1.2 punts)

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;
}
```

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

0.5 p. (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).

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

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

**Qüestió 4** (1 punt)

En un programa paral · lel es disposa d'un vector distribuït per blocs entre els processos, de manera que cada procés té el seu bloc en l'array `vloc`.

Implementa una funció que desplace els elements del vector una posició a la dreta, fent a més que l'últim element passe a ocupar la primera posició. Per exemple, si tenim 3 processos, donat l'estat inicial:

| | $P_0$ | $P_1$ | $P_2$ |
|---|---|---|---|
| `vloc` | `[2 5 3]` | `[7 1 0]` | `[6 4 9]` |

L'estat final seria:

| | $P_0$ | $P_1$ | $P_2$ |
|---|---|---|---|
| `vloc` | `[9 2 5]` | `[3 7 1]` | `[0 6 4]` |

La funció hauria d'evitar possibles interbloquejos. La capçalera de la funció serà:

```
void despl(double vloc[], int mb)
```

on `mb` és el nombre d'elements de `vloc` (suposarem que `mb` $> 1$).

**Qüestió 5** (0.8 punts)

Implementa una funció en C per a enviar a tots els processos les tres diagonals principals d'una matriu, sense tindre en compte ni la primera ni l'última fila. Per exemple, per a a una matriu de grandària 6 s'haurien d'enviar els elements marcats amb `x`:

```
+ + + + + +
x x x + + +
+ x x x + +
+ + x x x +
+ + + x x x
+ + + + + +
```

S'ha de fer definint un nou tipus de dades MPI que permeta enviar el bloc tridiagonal indicat utilitzant un sol missatge. Recorda que en C les matrius s'emmagatzemen en memòria per files. Utilitza esta capçalera per a la funció:

```
void envia_tridiagonal(double A[N][N],int root,MPI_Comm comm)
```

on

- `N` es el nombre de files i columnes de la matriu.
- `A` és la matriu amb les dades a enviar (en el procés que envia les dades) i la matriu on s'hauran de rebre (en la resta de procesos).

- El paràmetre `root` indica quin procés té inicialment en la seua matriu `A` les dades que s'han d'enviar a la resta de processos.
- `comm` és el comunicador que engloba tots els processos que hauran d'acabar tenint la part tridiagonal de `A`.

Per exemple, si es fera la crida a la funció:

```
envia_tridiagonal(A,5,comm);
```

el procés 5 seria el que té les dades vàlides en `A` a l'entrada a la funció, i a l'eixida tots els processos de `comm` haurien de tindre la part tridiagonal (menys la primera i última files).

**Qüestió 6** (1.2 punts)

El següent programa calcula el nombre d'elements per damunt de la mitja d'una matriu.

```
int PerDamuntDeMitja(double A[N][N], double mitja, int n, int m) {
  int i,j,nc=0;
  for (i=0;i<n;i++)
    for (j=0;j<m;j++)
      if (A[i][j]>mitja) nc++;
  return nc;
}
double Suma(double A[N][N], int n, int m) {
  int i,j;
  double suma=0.0;
  for (i=0;i<n;i++)
    for (j=0;j<m;j++)
      suma += A[i][j];
  return suma;
}
int main(int argc, char *argv[]) {
  int nc;
  double mitja, A[N][N];
  omplir(A,N,N);
  mitja = Suma(A,N,N);
  mitja /= N*N;
  printf("Mitja: %5.2f\n", mitja);
  nc = PerDamuntDeMitja(A,mitja,N,N);
  printf("Per damunt: %d\n",nc);
  return 0;
}
```

0.9 p. (a) Paral · lelitza el programa anterior mitjançant MPI, modificant únicament la funció `main`. Només el procés 0 haurà d'omplir la matriu i mostrar els missatges per pantalla. Es pot suposar que `N` és divisible entre el nombre de processos.

0.3 p. (b) Calcula el cost seqüencial i paral · lel, així com l'speed-up i el seu valor asimptòtic quan la grandària del problema tendeix a infinit. Nota: S'ha de considerar que la funció `omplir` no té cost computacional, i que el cost de comparar dos nombres reals és d'1 Flop.

**Computació Paral · lela**
Grau en Enginyeria Informàtica (ETSInf)
Curs 2018-19 ◇ Examen parcial 8/11/2018 ◇ Duració: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

**Qüestió 1** (1.25 punts)

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;
}
```

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

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

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

**Qüestió 2** (1.25 punts)

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;
   }
}
```

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

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

0.25 p. (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.

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

**Qüestió 3** (1 punt)

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);
}
```

**Computació Paral · lela**
Grau en Enginyeria Informàtica (ETSInf)
Curs 2018-19 ◇ Examen parcial 14/1/2019 ◇ Duració: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

**Qüestió 1** (1.3 punts)

En el següent programa seqüencial, on indiquem amb comentaris el cost computacional de cada funció, totes les funcions invocades modifiquen únicament el primer argument. Observeu que `A`, `D` i `E` són vectors, mentres que `B` i `C` són matrius.

```
#include <stdio.h>
int main (int argc, char *argv[]) {
    double A[N], B[N][N], C[N][N], D[N], E[N], res;
    read(A);              // T0, cost N
    generate(B,A);        // T1, cost 2N
    process2(C,B);        // T2, cost 2N^2
    process3(D,B);        // T3, cost 2N^2
    process4(E,C);        // T4, cost N^2
    res = process5(E,D); // T5, cost 2N
    printf("Result: %f\n", res);
    return 0;
}
```

0.2 p. (a) Obteniu el graf de dependències.

0.8 p. (b) Implementeu una versió paral · lela amb MPI, tenint en compte els següents aspectes:

- Utilitzeu el nombre més apropiat de processos paral · lels perquè l'execució siga el més ràpida possible, mostrant un missatge d'error en cas de que el nombre de processos en execució no coincidisca amb ell. Només el procés $P_0$ ha de realitzar les operacions `read` i `printf`.
- Presteu atenció a la grandària dels missatges i utilitzeu les tècniques d'agrupament i replicació, etc. si fora convenient.
- Realitzeu la implementació del programa complet.

0.3 p. (c) Calculeu el cost seqüencial, cost paral · lel, speed-up i eficiència.

**Qüestió 2** (1.2 punts)

Donada una matriu `A`, de `M` files i `N` columnes, la següent funció torna en el vector `sup` el nombre d'elements de cada fila que són superiors a la mitja.

```
void func(double A[M][N], int sup[M]) {
   int i, j;
   double mitja = 0;
   /* Calcula la mitja dels elements de A */
   for (i=0; i<M; i++)
      for (j=0; j<N; j++)
         mitja += A[i][j];
```

```
   mitja = mitja/(M*N);
   /* Conta nombre d'elements > mitja en cada fila */
   for (i=0; i<M; i++) {
      sup[i] = 0;
      for (j=0; j<N; j++)
         if (A[i][j]>mitja) sup[i]++;
   }
}
```

Escriviu una versió paral · lela de la funció anterior utilitzant MPI amb operacions de comunicació col · lectives, tenint en compte que la matriu `A` es troba inicialment en el procés 0, i que en finalitzar la funció el vector `sup` ha d'estar també en el procés 0. Els càlculs de la funció han de repartir-se de forma equitativa entre tots els processos. Es pot suposar que el nombre de files de la matriu és divisible entre el nombre de processos.

**Qüestió 3** (1 punt)

El següent fragment de codi MPI implementa un algoritme en el que cada procés calcula una matriu de `M` files i `N` columnes, i totes eixes matrius s'arrepleguen en el procés $P_0$ formant una matriu global de `M` files i `N*p` columnes (on `p` és el nombre de processos), de forma que les columnes de $P_1$ apareixen a continuació de les de $P_0$, després les de $P_2$, i així successivament.

```
int rank, i, j, k, p;
double alocal[M][N];
MPI_Comm_size(MPI_COMM_WORLD,&p);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
/* initialization of alocal omitted here */
if (rank==0) {
  double aglobal[M][N*p];
  /* copy part belonging to P0 */
  for (i=0;i<M;i++)
    for (j=0;j<N;j++)
      aglobal[i][j] = alocal[i][j];
  /* receive data from other processes */
  for (k=1;k<p;k++)
    for (i=0;i<M;i++)
      MPI_Recv(&aglobal[i][k*N],N,MPI_DOUBLE,k,33,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
  write(p,aglobal);
} else {
  for (i=0;i<M;i++)
    MPI_Send(&alocal[i][0],N,MPI_DOUBLE,0,33,MPI_COMM_WORLD);
}
```

0.8 p. (a) Modifiqueu el codi per tal que cada procés envie un únic missatge, en lloc d'un missatge per cada fila de la matriu. Per a fer-ho, s'haurà de definir un tipus de dades MPI per a la recepció.

0.2 p. (b) Calculeu el temps de comunicació tant de la versió original com de la modificada.

**Qüestió 1** (1 punt)

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;
}
```

0.6 p. (a) Paral · lelitza el codi anterior de forma eficient mitjançant OpenMP. De les possibles planificacions, quines podrien ser les més eficients? Justifica la resposta.

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

**Qüestió 2** (1.35 punts)

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;
}
```

0.1 p. (a) Calcula el temps d'execució seqüencial.

0.4 p. (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.

0.6 p. (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.

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

**Qüestió 3** (1.15 punts)

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);
}
```

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

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

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

**Qüestió 4** (1.2 punts)

Observa el següent programa, que llig un vector d'un fitxer, el modifica i mostra un resum per pantalla a més d'escriure el vector resultant en fitxer:

```
double facto(int m,double x)
{
  int i;
  double p = 1.0;
  for (i=1; i<=m; i++) {
    p = p * x;
    x = x + 1.0;
  }
  return p;
}
```

```
int main(int argc,char *argv[])
{
  int i, n;
  double a = 1.0, v[MAXN];

  n = llig_vector(v);
  for (i=0; i<n; i++) {
    v[i] = facto(n,v[i]);
    a = a * v[i];
  }
  printf("Factor alfalfa: %.2f\n",a);
  escriu_vector(n,v);
  return 0;
}
```

0.7 p. (a) Paral · lelitza'l amb MPI utilitzant operacions de comunicació col · lectiva allà on siga possible. L'entrada/eixida per pantalla i fitxer ha de fer-la només el procés 0. Se suposa que la grandària del vector (`n`) és un múltiple exacte del nombre de processos. Observa que la grandària del vector no és coneguda a priori, sinó que la retorna la funció `llig_vector`.

0.2 p. (b) Calculeu el temps d'execució seqüencial.

0.3 p. (c) Calcula el temps d'execució paral · lel, indicant clarament el temps de cada operació de comunicació. No simplifiques les expressions, deixa-ho indicat.

**Qüestió 5** (1 punt)

Es vol distribuir una matriu `A` de `F` files i `C` columnes entre els processos d'un comunicador MPI, utilitzant una distribució per blocs de columnes. El nombre de processos és `C`/2 on `C` és parell, de manera que la matriu local `Aloc` de cada procés tindrà 2 columnes.

Implementa una funció amb la següent capçalera que realitze la distribució esmentada, utilitzant comunicació punt a punt. La matriu `A` es troba inicialment en el procés 0, i en acabar la funció cada procés haurà de tindre en `Aloc` la part local que li corresponga de la matriu.

S'ha d'utilitzar el tipus de dades MPI adequat per a que només s'envie un missatge per procés.

```
void distrib(double A[F][C], double Aloc[F][2], MPI_Comm com)
```

**Qüestió 6** (1.3 punts)

Donada la següent funció, que calcula la suma d'un vector de `N` elements:

```
double suma(double v[N])
{
  int i;
  double s = 0.0;
  for (i=0; i<N; i++) s += v[i];
  return s;
}
```

0.7 p. (a) Paral · lelitza-la amb MPI utilitzant únicament comunicacions punt a punt. El vector `v` està inicialment només en el procés 0 i el resultat es vol que siga correcte en *tots* els processos. Se suposa que la grandària del vector (`N`) és un múltiple exacte del nombre de processos.

0.6 p. (b) Paral · lelitza-la amb MPI amb les mateixes premisses de l'apartat anterior, però ara utilitzant comunicacions col · lectives on siga convenient.

**Computació Paral · lela**
Grau en Enginyeria Informàtica (ETSInf)
Curs 2019-20 ◇ Examen parcial 30/10/2019 ◇ Duració: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

**Qüestió 1** (1.3 punts)

```
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 */
```

0.6 p. (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.

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

**Qüestió 2** (1 punt)

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]++;
  }
}
```

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

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

**Qüestió 3** (1.2 punts)

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;
}
```

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

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

0.2 p. (c) Calculeu el cost seqüencial del codi original.

0.2 p. (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.

**Computació Paral · lela**
Grau en Enginyeria Informàtica (ETSInf)
Curs 2019-20 ◇ Examen parcial 7/1/2020 ◇ Duració: 1h 50m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

**Qüestió 1** (1.1 punts)

Es vol paral · lelitzar el següent codi mitjançant MPI.

```
void calcular(int n, double x[], double y[], double z[]) {
  int i;
  double alpha, beta;

  /* Llegir els vector x, y, z, de dimensio n */
  llegir(n, x, y, z);          /* tasca 1 */

  normalitza(n,x);             /* tasca 2 */
  beta = obtindre(n,y);        /* tasca 3 */
  normalitza(n,z);             /* tasca 4 */

  /* tasca 5 */
  alpha = 0.0;
  for (i=0; i<n; i++)
    if (x[i] > 0.0) { alpha = alpha + beta*x[i]; }
    else { alpha = alpha + x[i]*x[i]; }

  /* tasca 6 */
  for (i=0; i<n; i++) z[i] = z[i] + alpha*y[i];
}
```

Suposem que es disposa de 3 processos, dels quals només un ha de cridar a la funció `llegir`. Es pot assumir que el valor de `n` està disponible en tots els processos. El resultat final (`z`) pot quedar emmagatzemat en un qualsevol dels 3 processos. La funció `llegir` modifica els tres vectors, la funció `normalitza` modifica el seu segon argument i la funció `obtindre` no modifica cap dels seus arguments.

0.25 p. (a) Dibuixeu el graf de dependències de les diferents tasques.

0.85 p. (b) Escriviu el codi MPI que resol el problema utilitzant una assignació que maximitze el paral · lelisme i minimitze el cost de comunicacions.

**Qüestió 2** (1.3 punts)

Es vol implementar en MPI l'enviament pel procés 0 (i recepció en la resta de processos) de la diagonal principal i l'antidiagonal d'una matriu $A$, emprant per a tal fi tipus de dades derivades (un per a cada tipus de diagonal) i la menor quantitat possible de missatges. Suposarem que:

- $N$ és una constant coneguda.
- Els elements de la diagonal principal són: $A_{0,0}$, $A_{1,1}$, $A_{2,2}$, …, $A_{N-1,N-1}$.

- Els elements de l'antidiagonal són: $A_{0,N-1}$, $A_{1,N-2}$, $A_{2,N-3}$, ..., $A_{N-1,0}$.
- Només el procés 0 té la matriu $A$ i enviarà les diagonals esmentades completes a la resta de processos.

Un exemple per a una matriu de grandària $N = 5$ seria: $A = \begin{pmatrix} * & & & & * \\ & * & & * & \\ & & * & & \\ & * & & * & \\ * & & & & * \end{pmatrix}$

0.6 p. (a) Completeu la següent funció, on els processos de l'1 en avant emmagatzemen sobre la matriu `A` les diagonals rebudes:

```
void sendrecv_diagonals(double A[N][N]) {
```

0.7 p. (b) Completeu esta altra funció, variant de l'anterior, on tots els processos (incloent el procés 0) emmagatzemaran sobre els vectors `prin` i `anti` les corresponents diagonals:

```
void sendrecv_diagonals(double A[N][N], double prin[N], double anti[N]) {
```

**Qüestió 3** (1.1 punts)

Observeu la següent funció, que compta el nombre d'aparicions d'un valor en una matriu i indica també la primera fila en la que apareix:

```
void search(double A[M][N], double x) {
  int i,j,first,count;
  first = M ; count = 0;
  for (i=0; i<M; i++)
    for (j=0; j<N; j++)
      if (A[i][j] == x) {
        count++;
        if (i < first) first = i;
      }
  printf("%g està %d vegades, la primera vegada en la fila %d.\n",x,count,first);
}
```

0.9 p. (a) Paral · lelitzeu-la mitjançant MPI repartint la matriu A entre tots els processos disponibles. Tant la matriu com el valor a buscar estan inicialment disponibles únicament en el procés `owner`. Se suposa que el nombre de files i columnes de la matriu és un múltiple exacte del nombre de processos. El `printf` que mostra el resultat per pantalla l'ha de fer només un procés.

Utilitzeu operacions de comunicació col · lectiva allà on siga possible.

Per a fer-ho, completeu esta funció:

```
void par_search(double A[M][N], double x, int owner) {
  double Aloc[M][N];
```

0.2 p. (b) Indiqueu el cost de comunicacions de cada operació de comunicació que heu utilitzat en l'apartat anterior. Suposeu una implementació bàsica de les comunicacions.

**Qüestió 1** (1.25 punts)

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;
}
```

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

0.2 p. (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*.

0.55 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.

**Qüestió 2** (1.25 punts)

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
```

0.3 p. (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.

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

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

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

**Qüestió 3** (1 punt)

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.

**Qüestió 4** (1.2 punts)

0.6 p. (a) El següent fragment de codi utilitza primitives de comunicació punt a punt per a un patró de comunicació que pot efectuar-se mitjançant una única operació col · lectiva.

```
#define TAG 999
int sz, rank;
double val,res,aux;
MPI_Comm comm=MPI_COMM_WORLD;
MPI_Status stat;
val = ...
MPI_Comm_size(comm, &sz);
if (sz==1) res = val;
else {
  MPI_Comm_rank(comm, &rank);
  if (rank==0) {
    MPI_Recv(&aux, 1, MPI_DOUBLE, rank+1, TAG, comm, &stat);
    res = aux + val;
  } else if (rank==sz-1) {
    MPI_Send(&val, 1, MPI_DOUBLE, rank-1, TAG, comm);
  } else {
    MPI_Recv(&aux, 1, MPI_DOUBLE, rank+1, TAG, comm, &stat);
    aux = aux + val;
    MPI_Send(&aux, 1, MPI_DOUBLE, rank-1, TAG, comm);
  }
}
```

Escriviu la crida a la primitiva MPI de comunicació col · lectiva equivalent, amb els arguments corresponents.

0.6 p. (b) Donada la següent crida a una primitiva de comunicació col · lectiva:

```
double val=...;
MPI_Bcast(&val, 1, MPI_DOUBLE, 0, comm);
```

Escriviu un fragment de codi equivalent (ha de realitzar la mateixa comunicació) però utilitzant únicament primitives de comunicació punt a punt.

**Qüestió 5** (1.3 punts)

Escriviu un programa paral · lel amb MPI en el qual el procés 0 llija una matriu de $M \times N$ nombres reals de disc (amb la funció `read_mat`) i eixa matriu vaja passant d'un procés a un altre fins a arribar a l'últim, que li la tornarà al procés 0. El programa haurà de medir el temps total d'execució, sense comptar la lectura de disc, i mostrar-lo per pantalla.

Utilitzeu esta capçalera per a la funció principal:

```
int main(int argc,char *argv[])
```

i teniu en compte que la funció de lectura de la matriu té esta capçalera:

```
void read_mat(double A[M][N]);
```

1 p. (a) Escriviu el programa demanat.

0.3 p. (b) Indiqueu el cost teòric total de les comunicacions.

**Qüestió 6** (1 punt)

Es vol repartir una matriu de $M$ files i $N$ columnes, que es troba en el procés 0, entre 4 processos mitjançant un repartiment per columnes cíclic. Com exemple es mostra el cas d'una matriu de 6 files i 8 columnes.

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 \\ 31 & 32 & 33 & 34 & 35 & 36 & 37 & 38 \\ 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 \\ 51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 \end{bmatrix}$$

Quedaria repartida de la següent forma:

$$P_0 = \begin{bmatrix} 1 & 5 \\ 11 & 15 \\ 21 & 25 \\ 31 & 35 \\ 41 & 45 \\ 51 & 55 \end{bmatrix}, \quad P_1 = \begin{bmatrix} 2 & 6 \\ 12 & 16 \\ 22 & 26 \\ 32 & 36 \\ 42 & 46 \\ 52 & 56 \end{bmatrix}, \quad P_2 = \begin{bmatrix} 3 & 7 \\ 13 & 17 \\ 23 & 27 \\ 33 & 37 \\ 43 & 47 \\ 53 & 57 \end{bmatrix}, \quad P_3 = \begin{bmatrix} 4 & 8 \\ 14 & 18 \\ 24 & 28 \\ 34 & 38 \\ 44 & 48 \\ 54 & 58 \end{bmatrix}$$

Implementeu una funció en MPI que realitze, mitjançant primitives punt a punt i de la forma més eficient possible, l'enviament i recepció de l'esmentada matriu. Nota: La recepció de la matriu haurà de fer-se en una matriu compacta (en `lmat`), com mostra l'exemple anterior. Nota: El nombre de columnes se suposa que es un múltiple exacte de 4 i es reparteix sempre entre 4 processos.

Per a la implementació es recomana utilitzar la següent capçalera:

```
int MPI_Reparte_col_cic(float mat[M][N], float lmat[M][N/4])
```