Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2012-13 ◇ Examen parcial 5/11/2012 ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (0.5 puntos)

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

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

Cuestión 2 (0.75 puntos)

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

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

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

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

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

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

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

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

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

Cuestión 3 (1.25 puntos)
Dada la siguiente función:

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

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

Las funciones `calcula_X` tienen como entrada los vectores que reciben como argumentos y con ellos modifican el vector X indicado. Cada función únicamente modifica el vector que aparece en su nombre. Por ejemplo, la función `calcula_u` utiliza los vectores `v`, `w` y `z` para realizar unos cálculos que guarda en el vector `u`, pero no modifica ni `v`, ni `w`, ni `z`.

Esto implica, por ejemplo, que las funciones `calcula_v`, `calcula_w` y `calcula_z` son independientes y podrían realizarse simultáneamente. Sin embargo, la función `calcula_u` necesita que hayan terminado las otras, porque usa los vectores que ellas rellenan (`v`,`w`,`z`).

0.2 p. (a) Dibuja el grafo de dependencias de las diferentes tareas.

0.75 p. (b) Paraleliza la función de forma eficiente.

0.3 p. (c) Si suponemos que el coste de todas las funciones `calcula_X` es el mismo y que el coste de los bucles posteriores es despreciable, ¿cuál sería el speedup máximo posible?

Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2012-13 ◇ Examen final 21/1/2013 ◇ Duración: 3h

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1.25 puntos)
Dada la siguiente función:

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

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

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

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

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

Cuestión 2 (0.75 puntos)
Dada la siguiente función:

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

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

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

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

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

Cuestión 3 (0.5 puntos)

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

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

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

Cuestión 4 (1 punto)

Se quiere paralelizar el siguiente código mediante MPI. Suponemos que se dispone 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);
```

Todas las funciones leen y modifican ambos argumentos, también los vectores. Suponemos que los vectores `a`, `b` y `c` están almacenados en P_0, P_1 y P_2, respectivamente, y son demasiado grandes para poder ser enviados eficientemente de un proceso a otro.

0.25 p. (a) Dibuja el grafo de dependencias de las diferentes tareas, indicando qué tarea se asigna a cada proceso.

0.75 p. (b) Escribe el código MPI que resuelve el problema.

Cuestión 5 (1 punto)

La ∞-norma de una matriz se define como el máximo de las sumas de los valores absolutos de los elementos de cada fila: $\max_{i=1..n} \left\{ \sum_{j=0}^{m-1} |a_{i,j}| \right\}$. El siguiente código secuencial implementa dicha operación para el caso de una matriz cuadrada.

```
#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) Implementa una versión paralela mediante MPI utilizando operaciones de comunicación colectiva en la medida de lo posible. Se puede asumir que el tamaño del problema es un múltiplo exacto del número de procesos. La matriz está inicialmente almacenada en P_0 y el resultado debe quedar también en P_0. Nota: se sugiere utilizar la siguiente cabecera para la función paralela, donde `Alocal` es una matriz que se supone ya reservada en memoria, y que puede ser utilizada por la función para almacenar la parte local de la matriz `A`.

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

0.25 p. (b) Obtén el coste computacional y de comunicaciones del algoritmo paralelo. Se puede asumir que la operación `fabs` tiene un coste despreciable, así como las comparaciones.

0.25 p. (c) Calcula el speed-up y la eficiencia cuando el tamaño del problema tiende a infinito.

Cuestión 6 (0.5 puntos)

Sea `A` un array bidimensional de números reales de doble precisión, de dimensión `N`×`N`. Define un tipo de datos derivado MPI que permita enviar una submatriz de tamaño 3×3. Por ejemplo, la submatriz que empieza en `A[0][0]` serían los elementos marcados con $\star$:

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

(a) Realiza las correspondientes llamadas para el envío desde P_0 y la recepción en P_1 del bloque de la figura.

(b) Indica qué habría que modificar en el código anterior para que el bloque enviado por P_0 sea el que empieza en la posición (0,3), y que se reciba en P_1 sobre el bloque que empieza en la posición (3,0).

Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2013-14 ◇ Examen parcial 11/11/2013 ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1 punto)

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

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

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

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

Cuestión 2 (0.8 puntos)

Dada la siguiente función:

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

0.4 p. (a) Paralelízala con OpenMP usando dos regiones paralelas.

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

Cuestión 3 (1.2 puntos)

Teniendo en cuenta la definición de las siguientes funciones:

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

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

se pretende paralelizar el siguiente código:

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

0.3 p. (a) Realiza una paralelización basada en los bucles.

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

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

Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2013-14 ◇ Examen parcial 13/1/2014 ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1.2 puntos)

El siguiente programa cuenta el número de ocurrencias de un valor en una matriz.

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

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

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

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

0.8 p. (a) Haz una versión paralela MPI del programa anterior, utilizando operaciones de comunicación colectiva cuando sea posible. La función `leer` deberá ser invocada solo por el proceso 0. Se puede asumir que `DIM` es divisible entre el número de procesos. Nota: hay que escribir el programa completo, incluyendo la declaración de las variables y las llamadas necesarias para iniciar y cerrar MPI.

0.4 p. (b) Calcula el tiempo de ejecución paralelo, suponiendo que el coste de comparar dos números reales es de 1 flop. Nota: para el coste de las comunicaciones, suponer una implementación sencilla de las operaciones colectivas.

Cuestión 2 (1 punto)

En un programa MPI, un vector x, de dimensión n, se encuentra distribuido de forma cíclica entre p procesos, y cada proceso guarda en el array `xloc` los elementos que le corresponden.

Implementa la siguiente función, de forma que al ser invocada por todos los procesos, realice las comunicaciones necesarias para que el proceso 0 recoja en el array `x` una copia del vector completo, con los elementos correctamente ordenados según el índice global.

Cada proceso del 1 al $p-1$ deberá enviar al proceso 0 todos sus elementos mediante un único mensaje.

```
void comunica_vec(double xloc[],int n,int p,int rank,double x[])
/* rank es el indice del proceso local              */
/* Esta funcion asume que n es múltiplo exacto de p */
```

Cuestión 3 (0.8 puntos)

El siguiente fragmento de código es incorrecto (desde el punto de vista semántico, no porque haya un error en los argumentos). Indica por qué y propón dos soluciones distintas.

```
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);
```

Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2013-14 ◇ Examen final 27/1/14 ◇ Bloque OpenMP ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1 punto)

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

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

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

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

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

Cuestión 2 (1 punto)

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

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

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

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

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

Cuestión 3 (1 punto)

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

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

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

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

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

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

Cuestión 4 (0.8 puntos)

Queremos medir la latencia de un anillo de `p` procesos en MPI, entendiendo por latencia el tiempo que tarda un mensaje de tamaño 0 en circular entre todos los procesos. Un anillo de `p` procesos MPI funciona de la siguiente manera: P_0 envía el mensaje a P_1, cuando éste lo recibe, lo reenvía a P_2, y así sucesivamente hasta que llega a P_{p-1} que lo enviará a P_0. Escribe un programa MPI que implemente este esquema de comunicación y muestre la latencia. Es recomendable hacer que el mensaje dé varias vueltas al anillo, y luego sacar el tiempo medio por vuelta, para obtener una medida más fiable.

Cuestión 5 (1.2 puntos)

El siguiente programa paralelo MPI debe calcular la suma de dos matrices A y B de dimensiones $M \times N$ utilizando una distribución cíclica de filas, suponiendo que el número de procesos `p` es divisor de M y teniendo en cuenta que P_0 tiene almacenadas inicialmente las matrices A y 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) leer(A,B);

/* (a) Reparto cíclico de filas de A y B */
/* (b) Cálculo local de Al+Bl */
/* (c) Recogida de resultados en el proceso 0 */

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

0.5 p. (a) Implementa el reparto cíclico de filas de las matrices A y B, siendo `Al` y `Bl` las matrices locales. Para realizar esta distribución debes o bien definir un nuevo tipo de dato de MPI o bien usar comunicaciones colectivas.

0.2 p. (b) Implementa el cálculo local de la suma `Al`+`Bl`, almacenando el resultado en `Al`.

0.5 p. (c) Escribe el código necesario para que P_0 almacene en `A` la matriz $A + B$. Para ello, P_0 debe recibir del resto de procesos las matrices locales `Al` obtenidas en el apartado anterior.

Cuestión 6 (1 punto)

Se quiere implementar el cálculo de la ∞-norma de una matriz cuadrada, que se obtiene como el máximo de las sumas de los valores absolutos de los elementos de cada fila, $\max_{i=0}^{n-1} \left\{ \sum_{j=0}^{n-1} |a_{i,j}| \right\}$. Para ello, se propone un esquema maestro-trabajadores. A continuación, se muestra la función

correspondiente al maestro (el proceso con identificador 0). La matriz se almacena por filas en un array uni-dimensional, y suponemos que es muy dispersa (tiene muchos ceros), por lo que el maestro envía únicamente los elementos no nulos (función `comprime`).

```
int comprime(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 maestro(double *A,int n)
{
  double buf[n];
  double norma=0.0,valor;
  int fila,completos=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 = comprime(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 (completos<n) {
    MPI_Recv(&valor, 1, MPI_DOUBLE, MPI_ANY_SOURCE, TAG_RESU,
             MPI_COMM_WORLD, &status);
    if (valor>norma) norma=valor;
    completos++;
    if (fila<n) {
      k = comprime(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 parte de los procesos trabajadores, completando la siguiente función:

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

Nota: Para el valor absoluto se puede usar

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

Recuerda que `MPI_Status` contiene, entre otros, los campos `MPI_SOURCE` y `MPI_TAG`.

Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2014-15 ◇ Examen parcial 3/11/2014 ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1 punto)
Dada la siguiente función:

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

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

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

0.1 p. (c) Determina el speedup y la eficiencia.

Cuestión 2 (1 punto)
Dado el siguiente fragmento de código:

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

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

0.4 p. (b) Paraleliza el código mediante OpenMP.

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

Cuestión 3 (1 punto)

Dada la siguiente función:

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

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

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

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

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

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

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

Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2014-15 ◇ Examen parcial 12/1/2015 ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1 punto)

La siguiente función muestra por pantalla el máximo de un vector `v` de `n` elementos y su posición:

```
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("Máximo: %f. Posición: %d\n", max, posmax);
}
```

Escribe una versión paralela MPI con la siguiente cabecera, donde los argumentos `rank` y `np` han sido obtenidos mediante `MPI_Comm_rank` y `MPI_Comm_size`, respectivamente.

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

La función debe asumir que el array `v` del proceso 0 contendrá inicialmente el vector, mientras que en el resto de procesos dicho array podrá usarse para almacenar la parte local que corresponda. Deberán comunicarse los datos necesarios de forma que el cálculo del máximo se reparta de forma equitativa entre todos los procesos. Finalmente, sólo el proceso 0 debe mostrar el mensaje por pantalla. Se deben utilizar operaciones de comunicación punto a punto (no colectivas). Nota: se puede asumir que `n` es múltiplo del número de procesos.

Cuestión 2 (1 punto)

0.6 p. (a) Implementa mediante comunicaciones colectivas una función en MPI que sume dos matrices cuadradas `a` y `b` y deje el resultado en `a`, teniendo en cuenta que las matrices `a` y `b` se encuentran almacenadas en la memoria del proceso P_0 y el resultado final también deberá estar en P_0. Supondremos que el número de filas de las matrices (`N`, constante) es divisible entre el número de procesos. La cabecera de la función es:

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

0.4 p. (b) Determina el tiempo paralelo, el speed-up y la eficiencia de la implementación detallada en el apartado anterior, indicando brevemente para su cálculo cómo se realizarían cada una de las operaciones colectivas (número de mensajes y tamaño de cada uno). Puedes asumir una implementación sencilla (no óptima) de estas operaciones de comunicación.

Cuestión 3 (1 punto)

Implementa una función que, dada una matriz A de $N \times N$ números reales y un índice k (entre 0 y $N-1$), haga que la fila k y la columna k de la matriz se comuniquen desde el proceso 0 al resto de procesos (sin comunicar ningún otro elemento de la matriz). La cabecera de la función sería:

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

Debes crear y usar un tipo de datos que represente una columna de la matriz. No es necesario que se envíen juntas la fila y la columna, se pueden enviar por separado.

Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2014-15 ◇ Examen final 21/1/15 ◇ Bloque OpenMP ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1 punto)

Dada esta función 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) Paralelízala eficientemente mediante directivas OpenMP (sin alterar el código ya existente).

0.3 p. (b) Indica el coste a priori en flops, tanto secuencial como paralelo (considerando sólo operaciones en coma flotante y asumiendo que una comparación implica una resta). ¿Cuál sería el speed-up y la eficiencia para p procesadores? Asumir que n es un múltiplo exacto de p.

Cuestión 2 (1.3 puntos)

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

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

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

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

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

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

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

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

Cuestión 3 (0.7 puntos)

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

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

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

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

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

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

Cuestión 4 (1 punto)

Dada la siguiente función, donde suponemos que las funciones `T1`, `T3` y `T4` tienen un coste de n y las funciones `T2` y `T5` de $2n$, siendo n un valor constante.

```
double ejemplo(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) Dibuja el grafo de dependencias y calcula el coste secuencial.

0.6 p. (b) Paralelízalo usando MPI con dos procesos. Ambos procesos invocan la función con el mismo valor de los argumentos `i`, `j` (no es necesario comunicarlos). El valor de retorno de la función debe ser correcto en el proceso 0 (no es necesario que esté en ambos procesos).

0.2 p. (c) Calcula el tiempo de ejecución paralelo (cálculo y comunicaciones) y el speedup con dos procesos.

Cuestión 5 (1 punto)

Dado el siguiente fragmento de código de un programa paralelo:

```
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);
...
```

Escribe el código necesario para definir un tipo de datos MPI que permita que una columna cualquiera de `A` se pueda enviar o recibir con un sólo mensaje. Utiliza el tipo de datos para hacer que el proceso 0 le envíe la columna `j` al proceso `proc`, quien la recibirá sobre el array `col`, le cambiará el signo a los elementos, y se la devolverá al proceso 0, que la recibirá de nuevo sobre la columna `j` de la matriz `A`.

Cuestión 6 (0.4 puntos)

El siguiente fragmento de código utiliza primitivas de comunicación punto a punto para un patrón de comunicación que puede efectuarse mediante una única operación colectiva.

```
#define TAG 999
int i, k, sz, rank;
float z[LNZ], zfull[LNFULL];   /* suponemos 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);
}
```

Escribe la llamada a la primitiva MPI de comunicación colectiva equivalente.

Cuestión 7 (0.6 puntos)

Dada la siguiente llamada a una primitiva de comunicación colectiva:

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

Escribe un fragmento de código equivalente (debe realizar la misma comunicación y operaciones aritméticas asociadas) pero utilizando únicamente primitivas de comunicación punto a punto.

Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2015-16 ◇ Examen parcial 9/11/2015 ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1.2 puntos)
Dada la siguiente función:

```
#define N 6000
#define PASOS 6

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

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

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

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

Cuestión 2 (1 punto)
La siguiente función procesa una serie de transferencias bancarias. Cada transferencia tiene una cuenta origen, una cuenta destino y una cantidad de dinero que se mueve de la cuenta origen a la cuenta destino. La función actualiza la cantidad de dinero de cada cuenta (array `saldos`) y además devuelve la cantidad máxima que se transfiere en una sola operación.

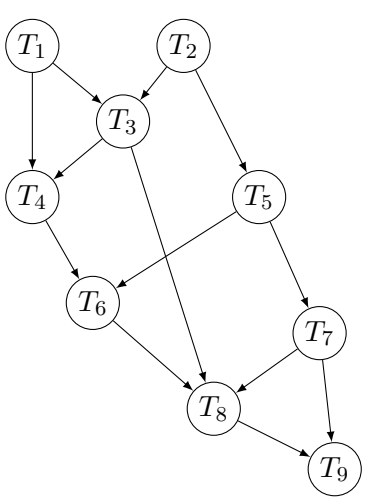
```
double transferencias(double saldos[], int origenes[],
       int destinos[], double cantidades[], int n)
{
   int i, i1, i2;
   double dinero, maxtransf=0;

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

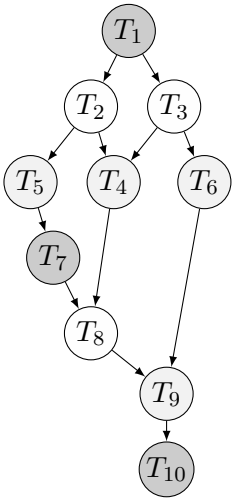
0.7 p. (a) Paraleliza la función de forma eficiente mediante OpenMP.

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

Cuestión 3 (0.8 puntos)

Dados los siguientes grafos de dependencias de tareas:





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

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

0.4 p.

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

Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2015-16 ◇ Examen parcial 15/1/2016 ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1 punto)

Se quiere implementar una función para distribuir una matriz cuadrada entre los procesos de un programa MPI, con la siguiente cabecera:

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

La matriz `A` se encuentra inicialmente en el proceso `root`, y debe distribuirse por filas entre los procesos, de manera que cada fila `i` debe ir al proceso `proc_fila[i]`. El contenido del array `proc_fila` es válido en todos los procesos. Cada proceso (incluido el `root`) debe almacenar las filas que le correspondan en la matriz local `Aloc`, ocupando las primeras filas (o sea, si a un proceso se le asignan `k` filas, éstas deben quedar almacenadas en las primeras `k` filas de `Aloc`).

Ejemplo para 3 procesos:

`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) Escribe el código de la función.

0.2 p. (b) En un caso general, ¿se podría usar el tipo de datos *vector* de MPI (`MPI_Type_vector`) para enviar a un proceso todas las filas que le tocan mediante un solo mensaje? Si se puede, escribe las instrucciones para definirlo. Si no se puede, justifica por qué.

Cuestión 2 (1 punto)

Esta función calcula el producto escalar de dos vectores:

```
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 función para realizar el producto escalar en paralelo mediante MPI, utilizando en la medida de lo posible operaciones colectivas. Se supone que los datos están disponibles en el proceso P_0 y que el resultado debe quedar también en P_0 (el valor de retorno de la función solo es necesario que sea correcto en P_0). Se puede asumir que el tamaño del problema `n` es exactamente divisible entre el número de procesos.

Nota: a continuación se muestra la cabecera de la función a implementar, incluyendo la declaración de los vectores locales (suponemos que `MAX` es suficientemente grande para cualquier valor de n y número de procesos).

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

0.3 p. (b) Calcula el speed-up. Si para un tamaño suficientemente grande de mensaje, el tiempo de envío por elemento fuera equivalente a 0.1 flops, ¿qué speed-up máximo se podría alcanzar cuando el tamaño del problema tiende a infinito y para un valor suficientemente grande de procesos?

0.2 p. (c) Modifica el código anterior para que el valor de retorno sea el correcto en todos los procesos.

Cuestión 3 (1 punto)

Se desea distribuir entre 4 procesos una matriz cuadrada de orden $2N$ ($2N$ filas por $2N$ columnas) definida a bloques como

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

donde cada bloque A_{ij} corresponde a una matriz cuadrada de orden N, de manera que se quiere que el proceso P_0 almacene localmente la matriz A_{00}, P_1 la matriz A_{01}, P_2 la matriz A_{10} y P_3 la matriz A_{11}.

Por ejemplo, la siguiente matriz con $N = 2$ quedaría distribuida como se muestra:

$$A = \left(\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array}\right) \qquad \begin{array}{ll} \text{En } P_0\text{:} \begin{pmatrix} 1 & 2 \\ 5 & 6 \end{pmatrix} & \text{En } P_1\text{:} \begin{pmatrix} 3 & 4 \\ 7 & 8 \end{pmatrix} \\ \text{En } P_2\text{:} \begin{pmatrix} 9 & 10 \\ 13 & 14 \end{pmatrix} & \text{En } P_3\text{:} \begin{pmatrix} 11 & 12 \\ 15 & 16 \end{pmatrix} \end{array}$$

0.8 p. (a) Implementa una función que realice la distribución mencionada, definiendo para ello el tipo de datos MPI necesario. La cabecera de la función sería:

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

donde `A` es la matriz inicial, almacenada en el proceso 0, y `B` es la matriz local donde cada proceso debe guardar el bloque que le corresponda de A.

Nota: se puede asumir que el número de procesos del comunicador es 4.

0.2 p. (b) Calcula el tiempo de comunicaciones.

Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2015-16 ◇ Examen final 26/1/16 ◇ Bloque OpenMP ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (0.8 puntos)
Dada la siguiente función:

```
double f1(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) Paralelízala con OpenMP de forma eficiente. Utiliza si es posible una sola región paralela.

0.2 p. (b) Haz que cada hilo muestre una línea con su identificador y el número de iteraciones del primer bucle `i` que ha realizado.

0.2 p. (c) En el primer bucle paralelo, justifica si cabe esperar diferencias en prestaciones entre las siguientes planificaciones para el bucle: `schedule(static)`, `schedule(static,1)`, `schedule(dynamic)`.

Cuestión 2 (1.2 puntos)
Dada la siguiente función:

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

```
  E(n,v);              /* Actualiza v, coste 3n   */
  F(n,w);              /* Actualiza w, coste 2n   */
  res = G(n,vec,v,w);  /* Calcula res, coste 3n   */
  return res;
}
```

0.5 p. (a) Dibuja el grafo de dependencias, indicando el grado máximo de concurrencia, la longitud del camino crítico y el grado medio de concurrencia.

0.5 p. (b) Paralelízala con OpenMP.

0.2 p. (c) Calcula el speedup y la eficiencia máximos si se ejecuta con 2 hilos.

Cuestión 3 (1 punto)

Sea la siguiente función:

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

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

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

Cuestión 4 (1 punto)

Sea el código secuencial:

```
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 versión paralela equivalente utilizando MPI, teniendo en cuenta los siguientes aspectos:

- El proceso P_0 obtiene inicialmente la matriz `A`, realizando la llamada `leer(A)`, siendo `leer` una función ya implementada.
- La matriz `A` se debe distribuir por bloques de filas entre todos los procesos.
- Finalmente P_0 debe contener el resultado en la matriz `A`.
- Utiliza comunicaciones colectivas siempre que sea posible.

Se supone que N es divisible entre el número de procesos y que la declaración de las matrices usadas es

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

0.2 p. (b) Calcula el speedup y la eficiencia.

Cuestión 5 (0.8 puntos)

Desarrolla una función que sirva para enviar una submatriz desde el proceso 0 al proceso 1, donde quedará almacenada en forma de vector. Se debe utilizar un nuevo tipo de datos, de forma que se utilice un único mensaje. Recuérdese que las matrices en C están almacenadas en memoria por filas.

La cabecera de la función será así:

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

Nota: se asume que `m*n<=MAX` y que la submatriz a enviar empieza en el elemento `A[0][0]`.

Ejemplo con $M = 4$, $N = 5$, $m = 3$, $n = 2$:

`A` (en P_0)						`v` (en P_1)
1	2	0	0	0		
3	4	0	0	0	$\rightarrow$	1 2 3 4 5 6
5	6	0	0	0		
0	0	0	0	0		

Cuestión 6 (1.2 puntos)

Dado el siguiente programa secuencial:

```
int calcula_valor(int vez);  /* función definida en otro sitio */

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

  printf("Número de iteraciones: ");
  scanf("%d",&n);
  min = 100000;
  for (i = 1; i <= n; i++) {
    val = calcula_valor(i);
    if (val < min) min = val;
  }
  printf("Mínimo: %d\n",min);
  return 0;
}
```

0.8 p. (a) Paralelízalo mediante MPI usando operaciones de comunicación punto a punto. La entrada y salida de datos debe hacerla únicamente el proceso 0. Puede asumirse que n es un múltiplo exacto del número de procesos.

0.4 p. (b) ¿Qué se cambiaría para usar operaciones de comunicación colectivas allá donde sea posible?

Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2016-17 ◇ Examen parcial 9/11/2016 ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1 punto)
Dada la siguiente función:

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

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

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

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

0.4 p. (b) Paraleliza la función con OpenMP.

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

Cuestión 2 (0.9 puntos)
Dada la siguiente función:

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

```
  for (j=0; j<N; j++) {     /* escalado de columnas */
    for (i=0; i<N; i++)
      A[i][j] *= s[j];
  }
}
```

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

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

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

Cuestión 3 (1.1 puntos)

La siguiente función calcula los puntos obtenidos por los distintos equipos de una liga de fútbol.

Durante la temporada ha habido `NJ` jornadas, en cada una de las cuales se han jugado `NPJ` partidos. La información de todos los partidos se encuentra almacenada en la matriz `partidos`, cada elemento de la cual es una estructura de datos (`SPartido`) que contiene los datos del partido (equipos que lo juegan y goles de cada uno).

Con esa información, la función calcula los `puntos` que le corresponden a cada uno de los `NE` equipos, teniendo en cuenta que un equipo consigue 3 puntos si gana un partido, 1 si lo empata, y cero si lo pierde.

Además, la función devuelve el número de goles de la temporada y obtiene también, para cada jornada, el máximo número de goles en un partido (array `maxg_jornada`).

```
int func(SPartido partidos[NJ][NPJ], int puntos[NE], int maxg_jornada[NJ]) {
   int i, j, maxg, eq1, eq2, g1, g2, goles_temporada;
   goles_temporada = 0;
   for (i=0; i<NJ; i++) {      /* para cada jornada */
      maxg = 0;
      for (j=0; j<NPJ; j++) {  /* para cada partido de la jornada */
         eq1 = partidos[i][j].eq1;
         eq2 = partidos[i][j].eq2;
         g1 = partidos[i][j].goles1;
         g2 = partidos[i][j].goles2;
         if (g1>g2)
            puntos[eq1] += 3;  /* Equipo 1 ha ganado */
         else if (g1<g2)
            puntos[eq2] += 3;  /* Equipo 2 ha ganado */
         else {
            puntos[eq1] += 1;  /* Empate */
            puntos[eq2] += 1;
         }
         if (g1+g2>maxg) maxg = g1+g2;
         goles_temporada += g1+g2;
      }
      maxg_jornada[i] = maxg;
   }
   return goles_temporada;
}
```

0.6 p. (a) Paraleliza (de forma eficiente) el bucle i.

0.5 p. (b) Paraleliza (de forma eficiente) el bucle `j`. Hay que tener en cuenta que en una misma jornada cada equipo juega un solo partido. Indica claramente los cambios respecto al código original (no respecto al apartado anterior).

Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2016-17 ◇ Examen parcial 16/1/2017 ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1 punto)

Se pretende distribuir con MPI los bloques cuadrados de la diagonal de una matriz cuadrada de dimensión $3 \cdot \mathrm{DIM}$ entre 3 procesos. Si la matriz fuera de dimensión 6 (DIM=2), la distribución sería como se ejemplifica:

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

Realiza una función que permita enviar los bloques con el mínimo número de mensajes. Se proporciona la definición de la cabecera de la función para facilitar la implementación. El proceso 0 tiene en `A` la matriz completa y tras la llamada a la función se debe tener en `Alcl` de cada proceso el bloque que le corresponde. Utiliza primitivas de comunicación punto a punto.

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

Cuestión 2 (1 punto)

El siguiente programa lee una matriz cuadrada A de orden N y construye, a partir de ella, un vector v de dimensión N de manera que su componente i-ésima, $0 \leq i < N$, es igual a la suma de los elementos de la fila i-ésima de la matriz A. Finalmente, el programa imprime 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) Utiliza comunicaciones colectivas para implementar un programa MPI que realice el mismo cálculo, de acuerdo con los siguientes pasos:

- El proceso P_0 lee la matriz A.
- P_0 reparte la matriz A entre todos los procesos.
- Cada proceso calcula la parte local de v.
- P_0 recoge el vector v a partir de las partes locales de todos los procesos.
- P_0 escribe el vector v.

Nota: Para simplificar, se puede asumir que N es divisible entre el número de procesos.

0.25 p. (b) Calcula los tiempos secuencial y paralelo, sin tener en cuenta las funciones de lectura y escritura. Indica el coste que has considerado para cada una de las operaciones colectivas realizadas.

Cuestión 3 (1 punto)

Dada la siguiente función, donde suponemos que las funciones `T1`, `T2` y `T3` tienen un coste de $7n$ y las funciones `T5` y `T6` de n, siendo n un valor constante.

```
double ejemplo(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) Dibuja el grafo de dependencias y calcula el coste secuencial.

0.5 p. (b) Paralelízala usando MPI, suponiendo que hay tres procesos. Todos los procesos invocan la función con el mismo valor del argumento `val` (no es necesario comunicarlo). El valor de retorno de la función debe ser correcto en el proceso 0 (no es necesario que lo sea en los demás procesos).

Nota: para las comunicaciones deben utilizarse únicamente operaciones de comunicación colectiva.

0.25 p. (c) Calcula el tiempo de ejecución paralelo (cálculo y comunicaciones) y el speedup con tres procesos. Obtén también el speedup asintótico, es decir, el límite cuando n tiene a infinito.

Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2016-17 ◇ Examen final 25/1/17 ◇ Bloque OpenMP ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (0.9 puntos)

```
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) Paraleliza con OpenMP (de forma eficiente) el bucle `i` de la función dada.

0.5 p. (b) Paraleliza con OpenMP (de forma eficiente) el bucle `j` de la función dada.

Cuestión 2 (1.1 puntos)

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

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

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

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

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

Cuestión 3 (1 punto)

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

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

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

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

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

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

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

Cuestión 4 (0.9 puntos)

Dada una matriz de `NF` filas y `NC` columnas, inicialmente almacenada en el proceso 0, se quiere hacer un reparto de la misma por bloques de columnas entre los procesos 0 y 1, de manera que las primeras NC/2 columnas se queden en el proceso 0 y el resto vayan al proceso 1 (supondremos que `NC` es par).

Implementa una función con la siguiente cabecera, que haga el reparto indicado mediante MPI, definiendo el tipo de datos necesario para que los elementos que le tocan al proceso 1 se transmitan mediante un solo mensaje. Cuando la función acabe, tanto el proceso 0 como el 1 deberán tener en `Aloc` el bloque de columnas que les toca. Puede que el número de procesos sea superior a 2, en cuyo caso sólo el 0 y el 1 deberán tener su bloque de matriz en `Aloc`.

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

Cuestión 5 (1.1 puntos)

Dada la siguiente función secuencial:

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

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

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

  return cont;
}
```

0.7 p. (a) Haz una versión paralela usando MPI, suponiendo que el vector `v` se encuentra inicialmente sólo en el proceso 0, y el resultado devuelto por la función sólo hace falta que sea correcto en el proceso 0. Deberán distribuirse los datos necesarios para que todos los cálculos se repartan de forma equitativa. Nota: Se puede asumir que `n` es divisible entre el número de procesos.

0.4 p. (b) Calcula el tiempo de ejecución de la versión paralela del apartado anterior, así como el límite del speedup cuando `n` tiende a infinito. Si has utilizado operaciones colectivas, indica cuál es el coste que has considerado para cada una de ellas.

Cuestión 6 (1 punto)

Desarrolla un programa *ping-pong*.

Se desea un programa paralelo, para ser ejecutado en 2 procesos, que repita 200 veces el envío del proceso 0 al proceso 1 y la devolución del proceso 1 al 0, de un mensaje de 100 enteros. Al final se deberá mostrar por pantalla el tiempo medio de envío de 1 entero, calculado a partir del tiempo de enviar/recibir todos estos mensajes.

El programa puede empezar así:

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

0.8 p. (a) Implementa el programa indicado.

0.2 p. (b) Calcula el tiempo de comunicaciones (teórico) del programa.

Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2017-18 ◇ Examen parcial 8/11/2017 ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1 punto)

La siguiente función calcula una matriz A a partir de un array de nodos (array `node`). Para cada nodo se calcula un coeficiente y con él se modifican tres elementos de A, dos en la diagonal y uno fuera de la diagonal.

```
double funcion(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 elementos en la diagonal de A */
    A[f][f] += coef;
    A[c][c] += coef;
    /* Modificar elemento fuera de la diagonal de A */
    A[f][c] -= coef;
    sum += coef;
  }
  return sum;
}
```

0.8 p. (a) Paralelízalo de forma eficiente usando OpenMP. Se sabe que dos nodos cualesquiera pueden modificar un mismo elemento de la diagonal, pero no un mismo elemento fuera de la diagonal. Justifica brevemente por qué lo haces así.

0.2 p. (b) Calcula el tiempo de ejecución a priori secuencial, suponiendo que la función `preparar` tiene un coste de n flops y `calculacoef` tiene un coste de 5 flops. Calcula el tiempo de ejecución paralelo y el speed-up. Calcula el speed-up máximo si se dispusiera de infinitos procesadores.

Cuestión 2 (0.8 puntos)

Dado el siguiente código:

```
int funcion(double A[M][N])
{
  int i,j,respuesta=1;
  double sumf,diag,elem;
  i = 0;
```

```
  while (i<M && respuesta==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) respuesta = 0;
    i++;
  }
  return respuesta;
}
```

0.5 p. (a) Indica cómo se paralelizaría mediante OpenMP, de la forma más eficiente, el bucle exterior (`while`).

0.3 p. (b) Paraleliza el segundo bucle (`for`).

Cuestión 3 (1.2 puntos)

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

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

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

0.4 p. (b) Paralelízala con OpenMP.

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

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

Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2017-18 ◇ Examen parcial 15/1/2018 ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1 punto)

Dada la siguiente función:

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

0.1 p. (a) Sabiendo que ninguna de las funciones llamadas modifica el vector `v`, dibuja el grafo de dependencias.

0.7 p. (b) Paralelízala con MPI utilizando únicamente operaciones de comunicación punto a punto. El vector `v` ya está replicado en todos los procesos. El resultado se necesita sólo en el proceso 0. Asegúrate de paralelizarlo de forma que no puedan producirse interbloqueos.

0.2 p. (c) Indica el tiempo teórico del algoritmo del apartado anterior (incluyendo también las comunicaciones).

Cuestión 2 (1.1 puntos)

El siguiente código proporciona el resultado de la operación $C = aA + bB$, siendo A, B y C matrices de $M \times N$ componentes y a y b números reales:

```
int main(int argc, char *argv[]) {
  int i, j;
  double a, b, A[M][N], B[M][N], C[M][N];
  LeeOperandos(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];
    }
  }
  EscribeMatriz(C);
  return 0;
}
```

Desarrolla una versión paralela mediante MPI utilizando operaciones colectivas, teniendo en cuenta que:

- P_0 obtendrá inicialmente las matrices A y B, así como los números reales a y b, tras invocar a la función `LeeOperandos`.
- Únicamente P_0 deberá disponer de la matriz C completa como resultado, y será el encargado de llamar a la función `EscribeMatriz`.
- M es un múltiplo exacto del número de procesos.
- Las matrices A y B se deberán distribuir cíclicamente por filas entre los procesos para llevar a cabo, en paralelo, la citada operación.

Cuestión 3 (0.9 puntos)

Se quiere implementar una operación de comunicación entre 3 procesos MPI en la que el proceso P_0 tiene almacenada una matriz `A` de dimensión $N \times N$, y debe enviar al proceso P_1 la submatriz formada por las filas de índice par y al proceso P_2 la submatriz formada por las filas de índice impar. Se deben utilizar tipos de datos derivados de MPI para realizar el menor número de envíos posibles. Cada matriz recogida en P_1 y P_2 deberá quedar almacenada en la matriz local `B` de dimensión $N/2 \times N$. Nota: se puede asumir que N es un número par.

Ejemplo: Si la matriz almacenada en P_0 es

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

la matriz recogida por P_1 debe ser

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

y la matriz recogida por P_2 debe ser

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

0.7 p. (a) Implementa para ello la función con la siguiente cabecera:

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

0.2 p. (b) Calcula el tiempo de comunicaciones.

Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2017-18 ◇ Examen final 25/1/18 ◇ Bloque OpenMP ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1.1 puntos)

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

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

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

0.5 p. (b) Paraleliza la función eficientemente con OpenMP.

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

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

Cuestión 2 (0.7 puntos)

Sea el siguiente código:

```
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);    /* Coste igual a N^3   */
genera2(B,b);    /* Coste 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;
   }
}
```

donde `genera1` y `genera2` son funciones previamente declaradas y definidas, de coste N^3 y $2N^3$, respectivamente. Paraleliza dicho código mediante OpenMP de la forma más eficiente posible usando solo una región paralela.

Cuestión 3 (1.2 puntos)

Dada la siguiente función:

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

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

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

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

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

Cuestión 4 (1 punto)

En un programa paralelo se dispone de un vector distribuido por bloques entre los procesos, de manera que cada proceso guarda su bloque en el array `vloc`.

Implementa una función que desplace los elementos del vector una posición a la derecha, haciendo además que el último elemento pase a ocupar la primera posición. Por ejemplo, si tenemos 3 procesos, dado el estado inicial:

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

El estado final sería:

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

La función deberá evitar los posibles interbloqueos. La cabecera de la función será:

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

donde `mb` es el número de elementos de `vloc` (supondremos que `mb` > 1).

Cuestión 5 (0.8 puntos)

Implementa una función en C para enviar a todos los procesos las tres diagonales principales de una matriz sin tener en cuenta ni la primera ni la última filas. Por ejemplo, para una matriz de tamaño 6 se tendrían que enviar los elementos marcados con `x`:

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

Debe hacerse definiendo un nuevo tipo de datos de MPI que permita enviar el bloque tridiagonal indicado utilizando un solo mensaje. Recuerda que en C las matrices se almacenan en memoria por filas. Utiliza la siguiente cabecera:

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

donde

- `N` es el número de filas y columnas de la matriz.
- `A` es la matriz con los datos a enviar (en el proceso que envía los datos) y la matriz donde se deberán recibir (en los demás procesos).

- El parámetro `root` indica qué proceso tiene inicialmente en su matriz `A` los datos que deben ser enviados a los demás procesos.
- `comm` es el comunicador que engloba todos los procesos que deberán acabar teniendo la parte tridiagonal de `A`.

Por ejemplo, si se llamara con

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

el proceso 5 sería el que tiene datos válidos en `A` a la entrada a la función, y a la salida todos los procesos de `comm` deberían tener la parte tridiagonal (menos la primera y última filas).

Cuestión 6 (1.2 puntos)

El siguiente programa calcula el número de elementos por encima de la media de una matriz.

```
int PorEncimaDeMedia(double A[N][N], double media, 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]>media) 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 media, A[N][N];
  rellena(A,N,N);
  media = Suma(A,N,N);
  media /= N*N;
  printf("Media: %5.2f\n", media);
  nc = PorEncimaDeMedia(A,media,N,N);
  printf("Por encima: %d\n",nc);
  return 0;
}
```

0.9 p. (a) Paraleliza el programa anterior mediante MPI, modificando únicamente la función `main`. Se asume que solo el proceso 0 debe rellenar la matriz y mostrar los mensajes por pantalla. Se puede asumir que `N` es divisible entre el número de procesos.

0.3 p. (b) Calcula el coste secuencial y paralelo, así como el speed-up y su valor asintótico cuando el tamaño del problema tiende a infinito. Nota: Considérese que la función `rellena` no tiene coste computacional y que el coste de comparar dos números reales es de 1 Flop.

Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2018-19 ◇ Examen parcial 8/11/2018 ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1.25 puntos)

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

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

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

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

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

Cuestión 2 (1.25 puntos)

Dada la siguiente función:

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

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

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

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

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

Cuestión 3 (1 punto)

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

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

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

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

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

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

Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2018-19 ◇ Examen parcial 14/1/2019 ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1.3 puntos)

En el siguiente programa secuencial, en el que indicamos con comentarios el coste computacional de cada función, todas las funciones invocadas modifican únicamente el primer argumento. Observa que `A`, `D` y `E` son vectores, mientras que `B` y `C` son matrices.

```
#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) Obtén el grafo de dependencias.

0.8 p. (b) Implementa una versión paralela con MPI, teniendo en cuenta los siguientes aspectos:

- Utiliza el número más apropiado de procesos paralelos para que la ejecución sea lo más rápida posible, mostrando un mensaje de error en caso de que el número de procesos en ejecución no coincida con éste. Solo el proceso P_0 debe realizar las operaciones `read` y `printf`.
- Presta atención al tamaño de los mensajes y utiliza las técnicas de agrupamiento y replicación, etc. si fuera conveniente.
- Realiza la implementación del programa completo.

0.3 p. (c) Calcula el coste secuencial, coste paralelo, speed-up y eficiencia.

Cuestión 2 (1.2 puntos)

Dada una matriz `A`, de `M` filas y `N` columnas, la siguiente función devuelve en el vector `sup` el número de elementos de cada fila que son superiores a la media.

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

```
   media = media/(M*N);
   /* Cuenta num. de elementos > media en cada fila */
   for (i=0; i<M; i++) {
      sup[i] = 0;
      for (j=0; j<N; j++)
         if (A[i][j]>media) sup[i]++;
   }
}
```

Escribe una versión paralela de la función anterior utilizando MPI con operaciones de comunicación colectivas, teniendo en cuenta que la matriz `A` se encuentra inicialmente en el proceso 0, y que al finalizar la función el vector `sup` debe estar también en el proceso 0. Los cálculos de la función deben repartirse de forma equitativa entre todos los procesos. Se puede suponer que el número de filas de la matriz es divisible entre el número de procesos.

Cuestión 3 (1 punto)

El siguiente fragmento de código MPI implementa un algoritmo en el que cada proceso calcula una matriz de `M` filas y `N` columnas, y todas estas matrices se recogen en el proceso P_0 formando una matriz global de `M` filas y `N*p` columnas (siendo `p` el número de procesos), de forma que las columnas de P_1 aparecen a continuación de las de P_0, luego las de P_2, y así sucesivamente.

```
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) Modifica el código para que cada proceso envíe un único mensaje, en lugar de un mensaje por cada fila de la matriz. Para ello, debes definir un tipo de datos MPI para la recepción.

0.2 p. (b) Calcula el tiempo de comunicación tanto de la versión original como de la modificada.

Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2018-19 ◇ Examen final 24/1/19 ◇ Bloque OpenMP ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1 punto)

Dada la siguiente función:

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

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

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

Cuestión 2 (1.35 puntos)

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

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

0.1 p. (a) Calcula el tiempo de ejecución secuencial.

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

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

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

Cuestión 3 (1.15 puntos)

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

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

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

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

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

Cuestión 4 (1.2 puntos)

Observa el siguiente programa, que lee un vector de un fichero, lo modifica y muestra un resumen por pantalla además de escribir el vector resultante en fichero:

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

0.7 p. (a) Paralelízalo con MPI usando operaciones de comunicación colectiva allá donde sea posible. La entrada/salida a pantalla y fichero debe hacerla únicamente el proceso 0. Asume que el tamaño del vector (`n`) es un múltiplo exacto del número de procesos. Observa que el tamaño del vector no es conocido a priori sino que lo devuelve la función `lee_vector`.

0.2 p. (b) Calcula el tiempo de ejecución secuencial.

0.3 p. (c) Calcula el tiempo de ejecución paralelo, indicando claramente el tiempo de cada operación de comunicación. No simplifiques las expresiones, déjalo indicado.

Cuestión 5 (1 punto)

Se quiere distribuir una matriz `A` de `F` filas y `C` columnas entre los procesos de un comunicador MPI, utilizando una distribución por bloques de columnas. El número de procesos es `C`/2 siendo `C` par, de manera que la matriz local `Aloc` de cada proceso tendrá 2 columnas.

Implementa una función con la siguiente cabecera que realice la distribución mencionada, utilizando comunicación punto a punto. La matriz `A` se encuentra inicialmente en el proceso 0, y al acabar la función cada proceso deberá tener en `Aloc` la parte local que le corresponda de la matriz.

Debe usarse el tipo de datos MPI adecuado para que solo se envíe un mensaje por proceso.

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

Cuestión 6 (1.3 puntos)

Dada la siguiente función, que calcula la suma de un vector de `N` elementos:

```
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) Paralelízala con MPI usando únicamente comunicaciones punto a punto. El vector `v` está inicialmente solo en el proceso 0 y el resultado se quiere correcto en *todos* los procesos. Puedes asumir que el tamaño del vector (`N`) es un múltiplo exacto del número de procesos.

0.6 p. (b) Paralelízala con MPI bajo las mismas premisas del apartado anterior, pero ahora usando comunicaciones colectivas donde sea conveniente.

Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2019-20 ◇ Examen parcial 30/10/2019 ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1.3 puntos)

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

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

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

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

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

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

Cuestión 2 (1 punto)

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

```
void histograma(int histo[], float notas[], int n) {
  int i, nota;
  float rnota;
  for (i=0;i<5;i++) histo[i] = 0;
  for (i=0;i<n;i++) {
```

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

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

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

Cuestión 3 (1.2 puntos)

Dada la siguiente función:

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

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

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

0.2 p. (c) Calcula el coste secuencial del código original.

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

Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2019-20 ◇ Examen parcial 7/1/2020 ◇ Duración: 1h 50m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1.1 puntos)

Se quiere paralelizar el siguiente código mediante MPI.

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

  /* Leer los vectores x, y, z, de dimension n */
  leer(n, x, y, z);           /* tarea 1 */

  normaliza(n,x);             /* tarea 2 */
  beta = obtener(n,y);        /* tarea 3 */
  normaliza(n,z);             /* tarea 4 */

  /* tarea 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]; }

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

Suponemos que se dispone de 3 procesos, de los cuales solo uno ha de llamar a la función `leer`. Se puede asumir que el valor de `n` está disponible en todos los procesos. El resultado final (`z`) puede quedar almacenado en uno cualquiera de los 3 procesos. La función `leer` modifica los tres vectores, la función `normaliza` modifica su segundo argumento y la función `obtener` no modifica ninguno de sus argumentos.

0.25 p. (a) Dibuja el grafo de dependencias de las diferentes tareas.

0.85 p. (b) Escribe el código MPI que resuelve el problema utilizando una asignación que maximice el paralelismo y minimice el coste de comunicaciones.

Cuestión 2 (1.3 puntos)

Se quiere implementar en MPI el envío por el proceso 0 (y recepción en el resto de procesos) de la diagonal principal y antidiagonal de una matriz A, empleando para ello tipos de datos derivados (uno para cada tipo de diagonal) y la menor cantidad posible de mensajes. Supondremos que:

- N es una constante conocida.
- Los elementos de la diagonal principal son: $A_{0,0}$, $A_{1,1}$, $A_{2,2}$, ..., $A_{N-1,N-1}$.

- Los elementos de la antidiagonal son: $A_{0,N-1}$, $A_{1,N-2}$, $A_{2,N-3}$, ..., $A_{N-1,0}$.
- Solo el proceso 0 posee la matriz A y enviará la totalidad de dichas diagonales al resto de procesos.

Un ejemplo para una matriz de tamaño $N = 5$ sería: $A = \begin{pmatrix} * & & & & * \\ & * & & * & \\ & & * & & \\ & * & & * & \\ * & & & & * \end{pmatrix}$

0.6 p. (a) Completa la siguiente función, donde los procesos del 1 en adelante almacenarán sobre la matriz `A` las diagonales recibidas:

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

0.7 p. (b) Completa esta otra función, variante de la anterior, donde todos los procesos (incluido el proceso 0) almacenarán sobre los vectores `prin` y `anti` las correspondientes diagonales:

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

Cuestión 3 (1.1 puntos)

Observa la siguiente función, que cuenta el número de apariciones de un número en una matriz e indica también la primera fila en la que aparece:

```
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 está %d veces, la primera vez en la fila %d.\n",x,count,first);
}
```

0.9 p. (a) Paralelízala mediante MPI repartiendo la matriz `A` entre todos los procesos disponibles. Tanto la matriz como el valor a buscar están inicialmente disponibles únicamente en el proceso `owner`. Asumimos que el número de filas y columnas de la matriz es un múltiplo exacto del número de procesos. El `printf` que muestra el resultado por pantalla debe hacerlo únicamente un proceso.

Utiliza operaciones de comunicación colectiva allí donde sea posible.

Para ello, completa esta función:

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

0.2 p. (b) Indica el coste de comunicaciones de cada operación de comunicación que has utilizado en el apartado anterior. Supón una implementación básica de las comunicaciones.

Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2019-20 ◇ Examen final 16/1/20 ◇ Bloque OpenMP ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1.25 puntos)
Dada la siguiente función:

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

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

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

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

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

Cuestión 2 (1.25 puntos)
Sea el siguiente código:

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

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

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

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

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

Cuestión 3 (1 punto)

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

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

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

Cuestión 4 (1.2 puntos)

0.6 p. (a) El siguiente fragmento de código utiliza primitivas de comunicación punto a punto para un patrón de comunicación que puede efectuarse mediante una única operación colectiva.

```
#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);
  }
}
```

Escribe la llamada a la primitiva MPI de comunicación colectiva equivalente, con los argumentos correspondientes.

0.6 p. (b) Dada la siguiente llamada a una primitiva de comunicación colectiva:

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

Escribe un fragmento de código equivalente (debe realizar la misma comunicación) pero utilizando únicamente primitivas de comunicación punto a punto.

Cuestión 5 (1.3 puntos)

Desarrolla un programa paralelo con MPI en el que el proceso 0 lea una matriz de $M \times N$ números reales de disco (con la función `read_mat`) y esta matriz se vaya pasando de un proceso a otro hasta llegar al último, que se la devolverá al proceso 0. El programa deberá medir el tiempo total de ejecución, sin contar la lectura de disco, y mostrarlo por pantalla.

Utiliza esta cabecera para la función principal:

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

y ten en cuenta que la función de lectura de la matriz tiene esta cabecera:

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

1 p. (a) Desarrolla el programa pedido.

0.3 p. (b) Indica el coste teórico total de las comunicaciones.

Cuestión 6 (1 punto)

Queremos repartir una matriz de M filas y N columnas que se encuentra en el proceso 0 entre 4 procesos mediante un reparto por columnas cíclico. Como ejemplo se muestra el caso de una matriz de 6 filas y 8 columnas.

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

Quedaría repartida de la siguiente forma:

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

Implementa una función en MPI que realice, mediante primitivas punto a punto y de la forma más eficiente posible, el envío y recepción de dicha matriz. Nota: La recepción de la matriz deberá hacerse en una matriz compacta (en `lmat`), como muestra el ejemplo anterior. Nota: El número de columnas se asume que es un múltiplo de 4 y se reparte siempre entre 4 procesos.

Para la implementación se recomienda utilizar la siguiente cabecera:

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