1. Comunicación punto a punto

Cuestión 1–1

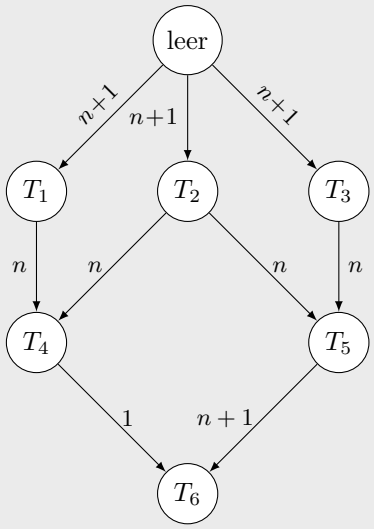
Dada la siguiente función:

```
double funcion()
{
  int i,n,j;
  double *v,*w,*z,sv,sw,x,res=0.0;

  /* Leer los vectores v, w, z, de dimension n */
  leer(&n, &v, &w, &z);

  calcula_v(n,v);              /* tarea 1 */
  calcula_w(n,w);              /* tarea 2 */
  calcula_z(n,z);              /* tarea 3 */

  /* tarea 4 */
  for (j=0; j<n; j++) {
    sv = 0;
    for (i=0; i<n; i++) sv = sv + v[i]*w[i];
    for (i=0; i<n; i++) v[i]=sv*v[i];
  }

  /* tarea 5 */
  for (j=0; j<n; j++) {
    sw = 0;
    for (i=0; i<n; i++) sw = sw + w[i]*z[i];
    for (i=0; i<n; i++) z[i]=sw*z[i];
  }

  /* tarea 6 */
  x = sv+sw;
  for (i=0; i<n; i++) res = res+x*z[i];
  return res;
}
```

Las funciones `calcula_X` tienen como entrada los datos que reciben como argumentos y con ellos modifican el vector `X` indicado. Por ejemplo, `calcula_v(n,v)` toma como datos de entrada los valores de `n` y `v` y modifica el vector `v`.

(a) Dibuja el grafo de dependencias de las diferentes tareas, incluyendo en el mismo el coste de cada una de las tareas y el volumen de las comunicaciones. Suponer que las funciones `calcula_X` tienen un coste de $2n^2$.

Solución: Los costes de comunicaciones aparecen en las aristas del grafo.



El coste (tiempo de ejecución) de la tarea 4 es:

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

El coste de T_5 es igual al de T_4, y el coste de T_6 es $2n$.

(b) Paralelízalo usando MPI, de forma que los procesos MPI disponibles ejecutan las diferentes tareas (sin dividirlas en subtareas). Se puede suponer que hay al menos 3 procesos.

Solución: Hay distintas posibilidades de hacer la asignación. Hay que tener en cuenta que sólo uno de los procesos debe realizar la lectura. Las tareas 1, 2 y 3 son independientes y por tanto se pueden asignar a 3 procesos distintos. Lo mismo ocurre con las 4 y 5.

Por ejemplo, el proceso 0 se puede encargar de leer, y hacer las tareas 1 y 4. El proceso 1 puede hacer la tarea 2, y el proceso 2 las tareas 3, 5 y 6.

Dado que los procesos distintos de P_0 no conocen el valor de `n`, es necesario enviar dicho valor en un mensaje separado, y tras su recepción ya se puede reservar la memoria necesaria con `malloc`. Este mensaje es de una longitud menor, ya que se envía una variable entera en lugar de tipo `double`. Sin embargo, por simplificar, no se ha considerado esta diferencia en el cálculo de costes.

```
double funcion()
{
  int i,n,j;
  double *v,*w,*z,sv,sw,x,res=0.0;
  int p,rank;
  MPI_Status st;

  MPI_Comm_size(MPI_COMM_WORLD,&p);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if (rank==0) {
    /* T0: Leer los vectores v, w, z, de dimension n */
    leer(&n, &v, &w, &z);
    MPI_Send(&n, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
```

```
      MPI_Send(&n, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
      MPI_Send(w, n, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
      MPI_Send(z, n, MPI_DOUBLE, 2, 0, MPI_COMM_WORLD);
      calcula_v(n,v);              /* tarea 1 */
      MPI_Recv(w, n, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &st);
      /* tarea 4 (mismo codigo del caso secuencial) */
      ...
      MPI_Send(&sv, 1, MPI_DOUBLE, 2, 0, MPI_COMM_WORLD);
      MPI_Recv(&res, 1, MPI_DOUBLE, 2, 0,MPI_COMM_WORLD,&st);
    }
    else if (rank==1) {
      MPI_Recv(&n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &st);
      w = (double*) malloc(n*sizeof(double));
      MPI_Recv(w, n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &st);
      calcula_w(n,w);              /* tarea 2 */
      MPI_Send(w, n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
      MPI_Send(w, n, MPI_DOUBLE, 2, 0, MPI_COMM_WORLD);
      MPI_Recv(&res, 1, MPI_DOUBLE, 2, 0,MPI_COMM_WORLD,&st);
    }
    else if (rank==2) {
      MPI_Recv(&n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &st);
      z = (double*) malloc(n*sizeof(double));
      MPI_Recv(z, n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &st);
      calcula_z(n,z);              /* tarea 3 */
      MPI_Recv(w, n, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &st);
      /* tarea 5 (mismo codigo del caso secuencial) */
      ...
      MPI_Recv(&sv, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,&st);
      /* tarea 6 (mismo codigo del caso secuencial) */
      ...
      /* Enviar resultado de tarea 6 a los demas procesos */
      MPI_Send(&res, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
      MPI_Send(&res, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
    }
    return res;
  }
```

(c) Indica el tiempo de ejecución del algoritmo secuencial, el del algoritmo paralelo, y el speedup que se obtendría. Ignorar el coste de la lectura de los vectores.

Solución: Teniendo en cuenta que el tiempo de ejecución de cada una de las tareas 1, 2 y 3 es de $2n^2$, mientras que el de las tareas 4 y 5 es de $3n^2$, y el de la tarea 6 es de $2n$, el tiempo de ejecución secuencial será la suma de esos tiempos:

$$t(n) = 3 \cdot 2n^2 + 2 \cdot 3n^2 + 2n \approx 12n^2$$

Tiempo de ejecución paralelo: tiempo aritmético. Será igual al tiempo aritmético del proceso que más operaciones realice, que en este caso es el proceso 2, que realiza las tareas 3, 5 y 6. Por tanto

$$t_a(n,p) = 2n^2 + 3n^2 + 2n \approx 5n^2$$

Tiempo de ejecución paralelo: tiempo de comunicaciones. Los mensajes que se transmiten son:

- 2 mensajes, del proceso 0 a los demás, con el valor de `n`. Coste de $2(t_s + t_w)$.

- 2 mensajes, uno del proceso 0 al 1 con el vector `w` y otro del 0 al 2 con el vector `z`. Coste de $2(t_s + nt_w)$.
- 2 mensajes, del proceso 1 a los demás, con el vector `w`. Coste de $2(t_s + nt_w)$.
- 1 mensaje, del proceso 0 al 2, con el valor de `sv`. Coste de $(t_s + t_w)$
- 2 mensajes, del proceso 2 a los demás, con el valor de `res`. Coste de $2(t_s + t_w)$

Por lo tanto, el coste de comunicaciones será:

$$t_c(n,p) = 5(t_s + t_w) + 4(t_s + nt_w) = 9t_s + (5 + 4n)t_w \approx 9t_s + 4nt_w$$

Tiempo de ejecución paralelo total:

$$t(n,p) = t_a(n,p) + t_c(n,p) = 5n^2 + 9t_s + 4nt_w$$

Speedup:

$$S(n,p) = \frac{12n^2}{5n^2 + 9t_s + 4nt_w}$$

Cuestión 1–2

Implementa una función que, a partir de un vector de dimensión `n`, distribuido entre `p` procesos de forma cíclica por bloques, realice las comunicaciones necesarias para que todos los procesos acaben con una copia del vector completo. Nota: utiliza únicamente comunicación punto a punto.

La cabecera de la función será:

```
void comunica_vector(double vloc[],int n,int b,int p,double w[])
/* vloc: parte local del vector v inicial
   n: dimension global del vector v
   b: tamaño de bloque empleado en la distribucion del vector v
   p: numero de procesos
   w: vector de longitud n, donde debe guardarse una copia de v
 */
```

Solución: Asumimos que `n` es múltiplo del tamaño de bloque `b` (o sea, todos los bloques tienen tamaño `b`).

```
void comunica_vector(double vloc[],int n,int b,int p,double w[])
/* vloc: parte local del vector v inicial
   n: dimension global del vector v
   b: tamaño de bloque empleado en la distribucion del vector v
   p: numero de procesos
   w: vector de longitud n, donde debe guardarse una copia de v
 */
{
  int i, rank, rank_pb, rank2;
  int ib, ib_loc;       /* Indice de bloque */
  int num_blq=n/b;      /* Numero de bloques */
  MPI_Status status;

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
  for (ib=0; ib<num_blq; ib++) {
    rank_pb = ib%p;           /* propietario del bloque */
    if (rank==rank_pb) {
      ib_loc = ib/p;         /* indice local del bloque */
      /* Enviar bloque ib a todos los procesos menos a mi */
      for (rank2=0; rank2<p; rank2++) {
        if (rank!=rank2) {
          MPI_Send(&vloc[ib_loc*b], b, MPI_DOUBLE, rank2, 0,
                   MPI_COMM_WORLD);
        }
      }
      /* Copiar bloque ib en mi propio vector local */
      for (i=0; i<b; i++) {
        w[ib*b+i]=vloc[ib_loc*b+i];
      }
    }
    else {
      MPI_Recv(&w[ib*b], b, MPI_DOUBLE, rank_pb, 0,
               MPI_COMM_WORLD, &status);
    }
  }
}
```

Cuestión 1–3

Se desea aplicar un conjunto de T tareas sobre un vector de números reales de tamaño `n`. Estas tareas han de aplicarse secuencialmente y en orden. La función que las representa tiene la siguiente cabecera:

```
void tarea(int tipo_tarea, int n, double *v);
```

donde `tipo_tarea` identifica el número de tarea de 1 hasta T. Sin embargo, estas tareas serán aplicadas a m vectores. Estos vectores están almacenados en una matriz A en el proceso maestro donde cada fila representa uno de esos m vectores.

Implementar un programa paralelo en MPI en forma de *Pipeline* donde cada proceso ($P_1 \dots P_{p-1}$) ejecutará una de las T tareas ($T = p - 1$). El proceso maestro (P_0) se limitará a alimentar el pipeline y recoger cada uno de los vectores (y almacenarlos de nuevo en la matriz A) una vez hayan pasado por toda la tubería. Utilizad un mensaje vacio identificado mediante una etiqueta para terminar el programa (supóngase que los esclavos desconocen el número m de vectores).

Solución: La parte de código que puede servir para implementar el pipeline propuesto podría ser la siguiente:

```
#define TAREA_TAG  123
#define FIN_TAG    1
int continuar,num;
MPI_Status stat;
if (!rank) {
  for (i=0;i<m;i++) {
    MPI_Send(&A[i*n], n, MPI_DOUBLE, 1, TAREA_TAG, MPI_COMM_WORLD);
  }
  MPI_Send(0, 0, MPI_DOUBLE, 1, FIN_TAG, MPI_COMM_WORLD);
  for (i=0;i<m;i++) {
```

```
      MPI_Recv(&A[i*n], n, MPI_DOUBLE, p-1, TAREA_TAG, MPI_COMM_WORLD, &stat);
    }
    MPI_Recv(0, 0, MPI_DOUBLE, p-1, FIN_TAG, MPI_COMM_WORLD, &stat);
  } else {
    continuar = 1;
    while (continuar) {
      MPI_Recv(A, n, MPI_DOUBLE, rank-1, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
      MPI_Get_count(&stat, MPI_DOUBLE, &num);
      if (stat.MPI_TAG == TAREA_TAG) {
        tarea(rank, n, A);
      } else {
        continuar = 0;
      }
      MPI_Send(A, num, MPI_DOUBLE, (rank+1)%p, stat.MPI_TAG, MPI_COMM_WORLD);
    }
  }
```

Cuestión 1–4

En un programa paralelo ejecutado en p procesos, se tiene un vector x de dimensión n distribuido por bloques, y un vector y replicado en todos los procesos. Implementar la siguiente función, la cual debe sumar la parte local del vector x con la parte correspondiente del vector y, dejando el resultado en un vector local z.

```
void suma(double xloc[],double y[],double z[],int n,int p,int pr)
/* pr es el indice del proceso local */
```

Solución:

```
void suma(double xloc[],double y[],double z[],int n,int p,int pr)
/* pr es el indice del proceso local */
{
  int i, iloc, mb;

  mb = ceil(((double) n)/p);
  for (i=pr*mb; i<MIN((pr+1)*mb,n); i++) {
    iloc=i%mb;
    z[iloc]=xloc[iloc]+y[i];
  }
}
```

Cuestión 1–5

La distancia de Levenshtein proporciona una medida de similitud entre dos cadenas. El siguiente código secuencial utiliza dicha distancia para calcular la posición en la que una subcadena es más similar a otra cadena, asumiendo que las cadenas se leen desde un fichero de texto.

Ejemplo: si la cadena `ref` contiene `"aafsdluqhqwBANANAqewrqerBANAfqrqerqrABANArqwrBAANANqwe"` y la cadena `str` contiene `"BANAN"`, el programa mostrará que la cadena `"BANAN"` se encuentra en la menor diferencia en la posición 11.

```
int mindist, pos, dist, i, ls, lr;
FILE *f1, *f2;
char ref[500], str[100];
```

```
f1 = fopen("ref.txt","r");
fgets(ref,500,f1);
lr = strlen(ref);
printf("Ref: %s (%d)\n", ref, lr);
fclose(f1);

f2 = fopen("lines.txt","r");
while (fgets(str,100,f2)!=NULL) {
  ls = strlen(str);
  printf("Str: %s (%d)\n", str, ls);
  mindist = levenshtein(str, ref);
  pos = 0;
  for (i=1;i<lr-ls;i++) {
    dist = levenshtein(str, &ref[i]);
    if (dist < mindist) {
      mindist = dist;
      pos = i;
    }
  }
  printf("Distancia %d para %s en %d\n", mindist, str, pos);
}
fclose(f2);
```

(a) Completa la siguiente implementación paralela MPI de dicho algoritmo según el modelo maestro-trabajadores:

```
int mindist, pos, dist, i, ls, lr, count, rank, size, rc, org;
FILE *f1, *f2;
char ref[500], str[100], c;
MPI_Status status;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

if (rank ==0) {    /* master */
  f1 = fopen("ref.txt","r");
  fgets(ref,500,f1);
  lr = strlen(ref);
  ref[lr-1]=0;
  lr--;
  MPI_Bcast(ref, lr+1, MPI_CHAR, 0, MPI_COMM_WORLD);
  printf("Ref: %s (%d)\n", ref, lr);
  fclose(f1);

  f2 = fopen("lines.txt","r");
  count = 1;
  while ( (fgets(str,100,f2)!=NULL) && (count<size) ) {
    ls = strlen(str);
    str[ls-1] = 0;
    ls--;
    MPI_Send(str, ls+1, MPI_CHAR, count, TAG_WORK, MPI_COMM_WORLD);
    count++;
```

```
  }

  do {
    printf("%d procesos activos\n", count);
    /*
      COMPLETAR
      - recibir tres mensajes del mismo proceso
      - leer nueva línea del fichero y enviarla
      - si fichero terminado, enviar mensaje de terminación
    */
  } while (count>1);

  fclose(f2);
} else {    /* worker */
    MPI_Bcast(ref, 500, MPI_CHAR, 0, MPI_COMM_WORLD);
    lr = strlen(ref);
    printf("[%d], Ref: %s\n", rank, ref);
    rc = 0;
    do {
      MPI_Recv(str, 100, MPI_CHAR, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      ls = strlen(str);
      if (status.MPI_TAG == TAG_WORK) {
        printf("[%d] Mensaje recibido (%s)\n", rank, str);
        mindist = levenshtein(str, ref);
        pos = 0;
        for (i=1;i<lr-ls;i++) {
          dist = levenshtein(str, &ref[i]);
          if (dist < mindist) {
            mindist = dist;
            pos = i;
          }
        }

        printf("[%d] envía: %d, %d, y %s a 0\n", rank, mindist, pos, str);
        MPI_Send(&mindist, 1, MPI_INT, 0, TAG_RESULT, MPI_COMM_WORLD);
        MPI_Send(&pos, 1, MPI_INT, 0, TAG_POS, MPI_COMM_WORLD);
        MPI_Send(str, ls+1, MPI_CHAR, 0, TAG_STR, MPI_COMM_WORLD);
      } else {
        printf("[%d] recibe mensaje con etiqueta %d\n", rank, status.MPI_TAG);
        rc = 1;
      }
   } while (!rc);
}
```

Solución:

```
   do {
     printf("%d procesos activos\n", count);
     MPI_Recv(&mindist, 1, MPI_INT, MPI_ANY_SOURCE, TAG_RESULT,
              MPI_COMM_WORLD, &status);
     org = status.MPI_SOURCE;
     MPI_Recv(&pos, 1, MPI_INT, org, TAG_POS, MPI_COMM_WORLD, &status);
```

```
        MPI_Recv(str, 100, MPI_CHAR, org, TAG_STR, MPI_COMM_WORLD, &status);
        ls = strlen(str);
        printf("De [%d]: Distancia %d para %s en %d\n", org, mindist, str, pos);
        count--;

        rc = (fgets(str,100,f2)!=NULL);
        if (rc) {
          ls = strlen(str);
          str[ls-1] = 0;
          ls--;
          MPI_Send(str, ls+1, MPI_CHAR, org, TAG_WORK, MPI_COMM_WORLD);
          count++;
        } else {
          printf("Enviando mensaje de terminación a %d\n", status.MPI_SOURCE);
          MPI_Send(&c, 1, MPI_CHAR, org, TAG_END, MPI_COMM_WORLD);
        }
      } while (count>1);
```

(b) Calcula el coste de comunicaciones de la versión paralela desarrollada dependiendo del tamaño del problema n y del número de procesos p.

Solución: En la versión propuesta, el coste de comunicación se debe a cuatro conceptos principales:

- Difusión de la referencia ($lr + 1$ bytes): $(t_s + t_w \cdot (lr + 1)) \cdot (p - 1)$
- Mensaje individual por cada secuencia ($ls_i + 1$ bytes): $(t_s + t_w \cdot (ls_i + 1)) \cdot n$
- Tres mensajes para la respuesta de cada secuencia (dos enteros más $ls_i + 1$ bytes): $(t_s + t_w \cdot (ls_i + 1)) \cdot n + 2 \cdot n \cdot (t_s + 4 \cdot t_w)$
- Mensaje de terminación (1 byte): $(t_s + t_w) \cdot (p - 1)$

Por tanto, el coste total se puede aproximar por $2 \cdot n \cdot t_s + t_w \cdot (9 \cdot n + m)$.

Cuestión 1–6

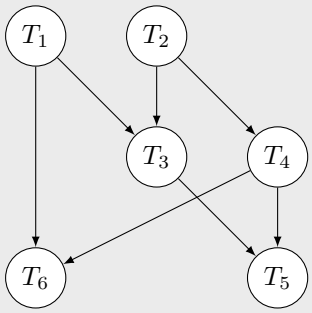
Se quiere paralelizar el siguiente código mediante MPI. Suponemos que se dispone de 3 procesos.

```
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.

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

Solución: El grafo de dependencias es el siguiente:



Debido a la restricción de dónde están situados los vectores, haremos la siguiente asignación: T_1 y T_6 en P_0, T_2 y T_3 en P_1, T_4 y T_5 en P_2.

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

Solución:

```
double a[N],b[N],c[N],v=0.0,w=0.0;
int p,rank;
MPI_Comm_size(MPI_COMM_WORLD,&p);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if (rank==0) {
  T1(a,&v);
  MPI_Send(&v, 1, MPI_DOUBLE, 1, 111, MPI_COMM_WORLD);
  MPI_Recv(&w, 1, MPI_DOUBLE, 2, 111, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
  T6(a,&w);
} else if (rank==1) {
  T2(b,&w);
  MPI_Send(&w, 1, MPI_DOUBLE, 2, 111, MPI_COMM_WORLD);
  MPI_Recv(&v, 1, MPI_DOUBLE, 0, 111, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
  T3(b,&v);
  MPI_Send(&v, 1, MPI_DOUBLE, 2, 111, MPI_COMM_WORLD);
} else { /* rank==2 */
  MPI_Recv(&w, 1, MPI_DOUBLE, 1, 111, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
  T4(c,&w);
  MPI_Send(&w, 1, MPI_DOUBLE, 0, 111, MPI_COMM_WORLD);
  MPI_Recv(&v, 1, MPI_DOUBLE, 1, 111, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
  T5(c,&v);
}
```

Cuestión 1–7

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);
```

Solución: El código realiza una comunicación en anillo, donde cada proceso envía `N` datos enteros al proceso de su derecha (y el último proceso envía al proceso 0). Es incorrecto porque se produce un interbloqueo, ya que la primitiva `MPI_Ssend` es síncrona y, por tanto, todos los procesos quedarán esperando a que se realice la recepción en el proceso destino. Sin embargo, ningún proceso alcanzará la primitiva `MPI_Recv`, por lo que la ejecución no progresará. El uso del envío estándar `MPI_Send` no resuelve el problema, ya que no se garantiza que no se haga también de forma síncrona.

Una solución sería implementar un protocolo pares-impares, reemplazando las dos últimas líneas por:

```
if (rank%2==0) {
  MPI_Ssend(sbuf, N, MPI_INT, dst, 111, MPI_COMM_WORLD);
  MPI_Recv(rbuf, N, MPI_INT, src, 111, MPI_COMM_WORLD, &stat);
} else {
  MPI_Recv(rbuf, N, MPI_INT, src, 111, MPI_COMM_WORLD, &stat);
  MPI_Ssend(sbuf, N, MPI_INT, dst, 111, MPI_COMM_WORLD);
}
```

Otra solución sería el uso de una primitiva combinada:

```
MPI_Sendrecv(sbuf, N, MPI_INT, dst, 111, rbuf, N, MPI_INT, src,
             111, MPI_COMM_WORLD, &stat);
```

También se resolvería utilizando primitivas no bloqueantes, por ejemplo:

```
MPI_Isend(sbuf, N, MPI_INT, dst, 111, MPI_COMM_WORLD, &req);
MPI_Recv(rbuf, N, MPI_INT, src, 111, MPI_COMM_WORLD, &stat);
MPI_Wait(&req, MPI_STATUS_IGNORE);
```

Cuestión 1–8

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`.

Solución:

```
void trabajador(int n)
{
  double buf[n];
  double s;
  int i,k;
  MPI_Status status;
  MPI_Recv(buf, n, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
           &status);
  while (status.MPI_TAG==TAG_FILA) {
    MPI_Get_count(&status, MPI_DOUBLE, &k);
    s=0.0;
    for (i=0;i<k;i++) s+=fabs(buf[i]);
    MPI_Send(&s, 1, MPI_DOUBLE, 0, TAG_RESU, MPI_COMM_WORLD);
    MPI_Recv(buf, n, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
           &status);
```

```
  }
}
```

Cuestión 1–9

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.

Solución:

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define REPS 1000

int main(int argc, char *argv[]) {
  int rank, i, size, prevp, nextp;
  double t1, t2;
  unsigned char msg;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  nextp = (rank+1)%size;
  if (rank>0) prevp = rank-1;
  else prevp = size-1;
  printf("Soy %d, mis vecinos son %d, %d\n", rank, prevp, nextp);

  t1 = MPI_Wtime();
  for (i=0;i<REPS;i++) {
    if (rank==0) {
      MPI_Send(&msg, 0, MPI_BYTE, nextp, i, MPI_COMM_WORLD);
      MPI_Recv(&msg, 0, MPI_BYTE, prevp, i, MPI_COMM_WORLD,
               MPI_STATUS_IGNORE);
    } else {
      MPI_Recv(&msg, 0, MPI_BYTE, prevp, i, MPI_COMM_WORLD,
               MPI_STATUS_IGNORE);
      MPI_Send(&msg, 0, MPI_BYTE, nextp, i, MPI_COMM_WORLD);
    }
  }
  t2 = MPI_Wtime();

  if (rank==0) {
    printf("Latencia con %d procesos: %f\n", size, (t2-t1)/REPS);
  }
  MPI_Finalize();
  return 0;
}
```

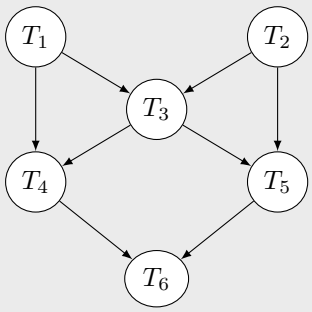
Cuestión 1–10

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 */
}
```

(a) Dibuja el grafo de dependencias y calcula el coste secuencial.

Solución: El grafo de dependencias se muestra en la siguiente figura:



El coste secuencial es: $t(n) = n + 2n + n + n + 2n + 4 \approx 7n$

Nota: los últimos 4 flops se refieren a las operaciones realizadas en la propia función `ejemplo` que no corresponden a ninguna de las funciones invocadas.

(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).

Solución: Para equilibrar la carga, proponemos una solución en la que el proceso 1 realiza `T2` y `T5`, y el resto de tareas se asignan al proceso 0.

```
double ejemplo(int i,int j)
{
  double a,b,c,d,e;
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if (rank==0) {
    a = T1(i);
    MPI_Recv(&b, 1, MPI_DOUBLE, 1, 111, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    c = T3(a+b,i);
    MPI_Send(&c, 1, MPI_DOUBLE, 1, 112, MPI_COMM_WORLD);
    d = T4(a/c);
    MPI_Recv(&e, 1, MPI_DOUBLE, 1, 113, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
```

```
    } else {
      b = T2(j);
      MPI_Send(&b, 1, MPI_DOUBLE, 0, 111, MPI_COMM_WORLD);
      MPI_Recv(&c, 1, MPI_DOUBLE, 0, 112, MPI_COMM_WORLD,
               MPI_STATUS_IGNORE);
      e = T5(b/c);
      MPI_Send(&e, 1, MPI_DOUBLE, 0, 113, MPI_COMM_WORLD);
    }
    return d+e;
  }
```

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

Solución: El tiempo paralelo de la implementación propuesta se debe calcular a partir del coste asociado al camino crítico del grafo de dependencias, correspondiente a $T_2 - T_3 - T_5 - T_6$.

$$t(n,2) = t_{arit}(n,2) + t_{comm}(n,2)$$

$$t_{arit}(n,2) = 2n + n + 2n + 3 \approx 5n$$

$$t_{comm}(n,2) = 3 \cdot (t_s + t_w)$$

$$t(n,2) = 5n + 3t_s + 3t_w$$

El speedup:

$$S(n,2) = \frac{t(n)}{t(n,2)} = \frac{7n}{5n + 3t_s + 3t_w}$$

Cuestión 1–11

Escribe una función con la siguiente cabecera, la cual debe hacer que los procesos con índices `proc1` y `proc2` intercambien el vector `x` que se pasa como argumento, mientras que en el resto de procesos el vector `x` no sufrirá ningún cambio.

```
void intercambiar(double x[N], int proc1, int proc2)
```

Debes tener en cuenta lo siguiente:

- Se debe evitar la posibilidad de interbloqueos.
- No se deben utilizar las funciones `MPI_Sendrecv`, `MPI_Sendrecv_replace` y `MPI_Bsend`.
- Declara las variables que consideres necesarias.
- Se supone que `N` es una constante definida previamente, y que `proc1` y `proc2` son índices de procesos válidos (en el rango entre 0 y el número de procesos menos uno).

Solución:

```
int rank;
double x2[N];
MPI_Status stat;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank==proc1) {
  MPI_Send(x, N, MPI_DOUBLE, proc2, 100, MPI_COMM_WORLD);
  MPI_Recv(x, N, MPI_DOUBLE, proc2, 100, MPI_COMM_WORLD, &stat);
```

```
} else if (rank==proc2) {
  int i;
  MPI_Recv(x2, N, MPI_DOUBLE, proc1, 100, MPI_COMM_WORLD, &stat);
  MPI_Send(x, N, MPI_DOUBLE, proc1, 100, MPI_COMM_WORLD);
  for (i=0; i<N; i++)
    x[i] = x2[i];
}
```

Cuestión 1–12

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.

Solución:

```
void func_par(double v[], int n, int rank, int np)
{
  double max, max2;
  int i, proc, posmax, posmax2, mb=n/np;

  if (rank==0)
    for (proc=1; proc<np; proc++)
      MPI_Send(&v[proc*mb], mb, MPI_DOUBLE, proc, 100,
               MPI_COMM_WORLD);
  else
    MPI_Recv(v, mb, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);

  max = v[0];
  posmax = 0;
```

```
    for (i=1; i<mb; i++) {
      if (v[i]>max) {
        max = v[i];
        posmax=i;
      }
    }
    posmax += rank*mb;    /* Convertir posmax en índice global */

    if (rank==0) {
      for (proc=1; proc<np; proc++) {
        MPI_Recv(&max2, 1, MPI_DOUBLE, proc, 100, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        MPI_Recv(&posmax2, 1, MPI_INT, proc, 100, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        if (max2>max) {
          max=max2;
          posmax=posmax2;
        }
      }
      printf("Maximo: %f. Posición: %d\n", max, posmax);
    } else {
      MPI_Send(&max, 1, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD);
      MPI_Send(&posmax, 1, MPI_INT, 0, 100, MPI_COMM_WORLD);
    }
  }
```

Cuestión 1–13

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

(a) Escribe el código de la función.

Solución:

```
void comunica(double A[][N], double Aloc[][N],
              int proc_fila[], int root)
{
  int i, j, iloc, rank;

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  iloc=0;
  for (i=0; i<N; i++) {
    /* Tratar la fila i */
    if (rank==root) {
      if (proc_fila[i]==root) {  /* Copia local de fila */
        for (j=0; j<N; j++) Aloc[iloc][j] = A[i][j];
        iloc++;
      }
      else
         MPI_Send(&A[i][0], N, MPI_DOUBLE, proc_fila[i], 0,
                  MPI_COMM_WORLD);
    }
    else if (rank==proc_fila[i]) {
      MPI_Recv(&Aloc[iloc][0], N, MPI_DOUBLE, root, 0,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      iloc++;
    }
  }
}
```

(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.

Solución: No se puede, porque las filas que le tocan a un proceso no tienen, en general, una separación constante entre ellas.

Cuestión 1–14

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];
```

(a) Implementa el programa indicado.

Solución:

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

```
#define V 200
#define N 100

int main(int argc,char *argv[])
{
  int i, yo;
  double t;
  int v[N];

  MPI_Init(&argc,&argv);

  MPI_Comm_rank(MPI_COMM_WORLD,&yo);

  t = MPI_Wtime();

  for (i=0;i<V;i++)
    if (yo==0) {
      MPI_Send(v, N, MPI_INT, 1, 20, MPI_COMM_WORLD);
      MPI_Recv(v, N, MPI_INT, 1, 17, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    } else {
      MPI_Recv(v, N, MPI_INT, 0, 20, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      MPI_Send(v, N, MPI_INT, 0, 17, MPI_COMM_WORLD);
    }

  t = MPI_Wtime() - t;
  if (yo==0) printf("Tiempo medio de envío de 1 entero: %g seg.\n",t/(2*V)/N);

  MPI_Finalize();
  return 0;
}
```

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

Solución:

$$t_c = 2 * V * (t_s + N * t_w) = 400(t_s + 100t_w)$$

Cuestión 1–15

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).

Solución:

```
void desplazar(double vloc[], int mb)
{
   int prev, sig, p, rank, i;
   double elem;
   MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if (rank==p-1) sig = 0;
   else sig = rank+1;
   if (rank==0) prev = p-1;
   else prev = rank-1;

   /* Desplazamiento local */
   elem = vloc[mb-1];
   for (i=mb-1; i>0; i--)
      vloc[i] = vloc[i-1];

   /* Comunicación con los vecinos */
   MPI_Sendrecv(&elem, 1, MPI_DOUBLE, sig, 0,
                &vloc[0], 1, MPI_DOUBLE, prev, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

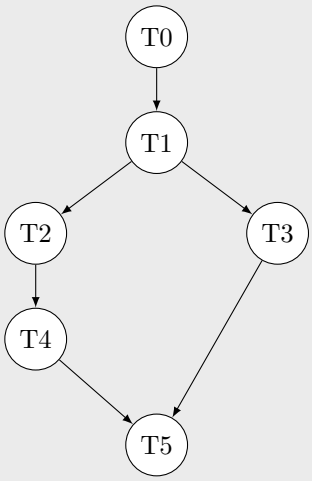
Cuestión 1–16

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;
}
```

(a) Obtén el grafo de dependencias.

Solución:



(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.

Solución: Utilizaremos dos procesos y aplicaremos replicación en la tarea `generate` ya que el coste de envío de la matriz `B` es superior al del envío del vector `A`, y la tarea `generate` no puede hacerse en paralelo con ninguna otra.

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
    double A[N], B[N][N], C[N][N], D[N], E[N], res;
    int rank, p;
    MPI_Status status;

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

    if (p==2) {
       if (rank==0) {
           read(A);             // T0, cost N
           MPI_Send(A,N,MPI_DOUBLE,1,0,MPI_COMM_WORLD);
       } else {
           MPI_Recv(A,N,MPI_DOUBLE,0,0,MPI_COMM_WORLD,&status);
       }
       generate(B,A);           // T1, cost 2N
       if (rank==0) {
           process2(C,B);       // T2, cost 2N^2
           process4(E,C);       // T4, cost N^2
           MPI_Recv(D,N,MPI_DOUBLE,1,0,MPI_COMM_WORLD,&status);
           res = process5(E,D); // T5, cost 2N
```

```
            printf("Result: %f\n", res);
          } else {
            process3(D,B);       // T3, cost 2N^2
            MPI_Send(D,N,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
          }
        } else
          if (rank==0)
            printf("Incorrect number of processes (%d)\n", p);

        MPI_Finalize();
        return 0;
    }
```

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

Solución:

$$t(n) = N + 2N + 2N^2 + 2N^2 + N^2 + 2N = 5N + 5N^2 \approx 5N^2 \text{ flops}$$

$$t(n,p) = N+(t_s+Nt_w)+2N+2N^2+N^2+(t_s+Nt_w)+2N = 5N+3N^2+2t_s+2Nt_w \approx 3N^2+2t_s+2Nt_w$$

$$S(n,p) = \frac{5N^2}{3N^2 + 2t_s + 2Nt_w}$$

$$E(n,p) = \frac{5N^2}{2(3N^2 + 2t_s + 2Nt_w)}$$

Cuestión 1–17

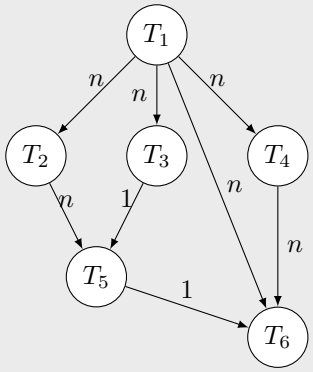
Se quiere paralelizar el siguiente código mediante MPI.

```
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.

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

Solución: El grafo de dependencias es el siguiente:



Aunque no se pide en la pregunta, el grafo muestra los arcos etiquetados con el volumen de datos que se transfiere entre cada par de tareas dependientes, lo que ha de tenerse en cuenta para seleccionar la mejor asignación de tareas a procesos posible.

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

Solución: Una asignación de tareas que cumple los requisitos es $P_0 : T_1, T_4, T_6$; $P_1 : T_3$; $P_2 : T_2, T_5$.

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

  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if (rank == 0) {
    /* Leer los vectores x, y, z, de dimension n */
    leer( n, x, y, z );      /* tarea 1 */
    MPI_Send(x, n, MPI_DOUBLE, 2, 123, MPI_COMM_WORLD);
    MPI_Send(y, n, MPI_DOUBLE, 1, 123, MPI_COMM_WORLD);
    normaliza(n,z);             /* tarea 4 */
    MPI_Recv(&alpha, 1, MPI_DOUBLE, 2, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    /* tarea 6 */
    for (i=0; i<n; i++) z[i] = z[i] + alpha*y[i];
  }
  else if (rank == 1) {
    MPI_Recv(y, n, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    beta = obtener(n,y);        /* tarea 3 */
    MPI_Send(&beta, 1, MPI_DOUBLE, 2, 123, MPI_COMM_WORLD);
  }
  else if (rank == 2) {
    MPI_Recv(x, n, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    normaliza(n,x);             /* tarea 2 */
    MPI_Recv(&beta, 1, MPI_DOUBLE, 1, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    /* 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]; }
      MPI_Send(&alpha, 1, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD );
    }
  }
```

Cuestión 1–18

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]);
```

(a) Desarrolla el programa pedido.

Solución:

```
#include <stdio.h>
#include <mpi.h>
#define M 1000
#define N 1000
int main(int argc,char *argv[]) {
  int id,np;
  double A[M][N],t1,t2;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&id);
  MPI_Comm_size(MPI_COMM_WORLD,&np);
  if (id==0) read_mat(A);
  t1=MPI_Wtime();
  if (id==0) {
    MPI_Send(A,M*N,MPI_DOUBLE,1,1234,MPI_COMM_WORLD);
    MPI_Recv(A,M*N,MPI_DOUBLE,np-1,1234,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
  } else {
    MPI_Recv(A,M*N,MPI_DOUBLE,id-1,1234,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    MPI_Send(A,M*N,MPI_DOUBLE,(id+1)%np,1234,MPI_COMM_WORLD);
  }
  t2=MPI_Wtime();
  if (id==0) printf("Tiempo: %.2f segundos.\n",t2-t1);
  MPI_Finalize();
  return 0;
}
```

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

Solución:

$$t = p \cdot (t_s + MNt_w)$$

2. Comunicación colectiva

Cuestión 2–1

El siguiente fragmento de código permite calcular el producto de una matriz cuadrada por un vector, ambos de dimensión `N`:

```
int i, j;
int A[N][N], v[N], x[N];
leer(A,v);
for (i=0;i<N;i++) {
  x[i]=0;
  for (j=0;j<N;j++) x[i] += A[i][j]*v[j];
}
escribir(x);
```

Escribe un programa MPI que realice el producto en paralelo, teniendo en cuenta que el proceso P_0 obtiene inicialmente la matriz `A` y el vector `v`, realiza una distribución de `A` por bloques de filas consecutivas sobre todos los procesos y envía `v` a todos. Asimismo, al final P_0 debe obtener el resultado. Nota: Para simplificar, se puede asumir que `N` es divisible por el número de procesos.

Solución: Definimos una matriz auxiliar `B` y un vector auxiliar `y`, que contendrán las porciones locales de `A` y `x` en cada proceso. Tanto `B` como `y` tienen `k=N/p` filas, pero para simplificar se han dimensionado a `N` filas ya que el valor de `k` es desconocido en tiempo de compilación (una solución eficiente en términos de memoria reservaría estas variables con `malloc`).

```
int i, j, k, rank, p;
int A[N][N], B[N][N], v[N], x[N], y[N];

MPI_Comm_size(MPI_COMM_WORLD,&p);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if (rank == 0) leer(A,v);
k = N/p;
MPI_Scatter(A, k*N, MPI_INT, B, k*N, MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(v, N, MPI_INT, 0, MPI_COMM_WORLD);
for (i=0;i<k;i++) {
  y[i]=0;
  for (j=0;j<N;j++) y[i] += B[i][j]*v[j];
}
MPI_Gather(y, k, MPI_INT, x, k, MPI_INT, 0, MPI_COMM_WORLD);
if (rank == 0) escribir(x);
```

Cuestión 2–2

El siguiente fragmento de código calcula la norma de Frobenius de una matriz cuadrada obtenida a partir de la función `leermat`.

```
int i, j;
double s, norm, A[N][N];
leermat(A);
s = 0.0;
for (i=0;i<N;i++) {
  for (j=0;j<N;j++) s += A[i][j]*A[i][j];
}
```

```
norm = sqrt(s);
printf("norm=%f\n",norm);
```

Implementa un programa paralelo usando MPI que calcule la norma de Frobenius, de manera que el proceso P_0 lea la matriz `A`, la reparta según una distribución cíclica de filas, y finalmente obtenga el resultado `s` y lo imprima en la pantalla. Nota: Para simplificar, se puede asumir que `N` es divisible por el número de procesos.

Solución: Utilizamos una matriz auxiliar `B` para que cada proceso almacene su parte local de `A` (sólo se utilizan las `k` primeras filas). Para la distribución de la matriz, se hacen `k` operaciones de reparto, una por cada bloque de `p` filas.

```
int i, j, k, rank, p;
double s, norm, A[N][N], B[N][N];

MPI_Comm_size(MPI_COMM_WORLD,&p);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
k = N/p;
if (rank == 0) leermat(A);
for (i=0;i<k;i++) {
  MPI_Scatter(&A[i*p][0],N,MPI_DOUBLE,&B[i][0],N,MPI_DOUBLE,0,
              MPI_COMM_WORLD);
}
s=0;
for (i=0;i<k;i++) {
  for (j=0;j<N;j++) s += B[i][j]*B[i][j];
}
MPI_Reduce(&s,&norm,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
if (rank == 0) {
  norm = sqrt(norm);
  printf("norm=%f\n",norm);
}
```

Cuestión 2–3

Se quiere paralelizar el siguiente programa usando MPI.

```
double *lee_datos(char *nombre, int *n) {
  ...  /* lectura desde fichero de los datos */
  /* devuelve un puntero a los datos y el número de datos en n */
}
double procesa(double x) {
  ...  /* función costosa que hace un cálculo dependiente de x */
}
int main() {
  int i,n;
  double *a,res;
  a = lee_datos("datos.txt",&n);
  res = 0.0;
  for (i=0; i<n; i++)
    res += procesa(a[i]);
  printf("Resultado: %.2f\n",res);
  free(a);
```

```
  return 0;
}
```

Aspectos a tener en cuenta:

- Sólo el proceso 0 debe llamar a `lee_datos` (sólo él leerá del fichero).
- Sólo el proceso 0 debe mostrar el resultado.
- Hay que repartir los `n` cálculos entre los procesos disponibles usando un reparto por bloques. Habrá que enviar a cada proceso su parte de `a` y recoger su aportación al resultado `res`. Se puede suponer que `n` es divisible por el número de procesos.

(a) Realiza una versión con comunicación punto a punto.

Solución:

```
int main(int argc,char *argv[])
{
  int i,n,p,np,nb;
  double *a,res,aux;
  MPI_Status stat;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&p);
  MPI_Comm_size(MPI_COMM_WORLD,&np);

  if (!p) a = lee_datos("datos.txt",&n);

  /* Difundir el tamaño del problema (1) */
  if (!p) {
    for (i=1; i<np; i++)
      MPI_Send(&n,1,MPI_INT,i,5,MPI_COMM_WORLD);
  } else {
    MPI_Recv(&n,1,MPI_INT,0,5,MPI_COMM_WORLD,&stat);
  }
  nb = n/np;  /* Asumimos que n es múltiplo de np */

  if (p) a = (double*) malloc(nb*sizeof(double));

  /* Repartir la a entre todos los procesos (2) */
  if (!p) {
    for (i=1; i<np; i++)
      MPI_Send(&a[i*nb],nb,MPI_DOUBLE,i,25,MPI_COMM_WORLD);
  } else {
    MPI_Recv(a,nb,MPI_DOUBLE,0,25,MPI_COMM_WORLD, &stat);
  }
  res = 0.0;
  for (i=0; i<nb; i++)
    res += procesa(a[i]);

  /* Recogida de resultados (3) */
  if (!p) {
    for (i=1; i<np; i++)
      MPI_Recv(&aux,1,MPI_DOUBLE,i,52,MPI_COMM_WORLD,&stat);
```

```
        res += aux;
      }
    } else {
      MPI_Send(&res,1,MPI_DOUBLE,0,52,MPI_COMM_WORLD);
    }
    if (!p) printf("Resultado: %.2f\n",res);
    free(a);

    MPI_Finalize();
    return 0;
  }
```

(b) Realiza una versión utilizando primitivas de comunicación colectiva.

Solución: Sólo hay que cambiar en el código anterior las zonas marcadas con (1), (2) y (3) por:

```
/* Difundir el tamaño del problema (1) */
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

/* Repartir la a entre todos los procesos (2) */
MPI_Scatter(a, nb, MPI_DOUBLE, b, nb, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);

/* Recogida de resultados (3) */
aux = res;
MPI_Reduce(&aux, &res, 1, MPI_DOUBLE, MPI_SUM, 0,
           MPI_COMM_WORLD);
```

En el *scatter* se ha utilizado una variable auxiliar `b`, ya que no está permitido usar el mismo buffer para envío y recepción (excepto en el caso especial en que se usa `MPI_IN_PLACE`). Faltaría cambiar `a` por `b` en las llamadas a `malloc`, `free` y `procesa`.

Cuestión 2–4

Desarrolla un programa MPI que juegue al siguiente juego:

1. Cada proceso se inventa un número y se lo comunica al resto.
2. Si todos los procesos han pensado el mismo número, se acaba el juego.
3. Si no, se repite el proceso (se vuelve a 1). Si ya ha habido 1000 repeticiones, se finaliza con un error.
4. Al final hay que indicar por pantalla (una sola vez) cuántas veces se ha tenido que repetir el proceso para que todos pensaran el mismo número.

Se dispone de la siguiente función para inventar los números:

```
int piensa_un_numero(); /* devuelve un número aleatorio */
```

Utiliza operaciones de comunicación colectiva de MPI para todas las comunicaciones necesarias.

Solución:

```
int p,np;
int num,*vnum,cont,iguales,i;

MPI_Comm_rank(MPI_COMM_WORLD,&p);
```

```
MPI_Comm_size(MPI_COMM_WORLD,&np);
vnum = (int*) malloc(np*sizeof(int));
cont = 0;
do {
  cont++;
  num = piensa_un_numero();
  MPI_Allgather(&num,1,MPI_INT,vnum,1,MPI_INT,MPI_COMM_WORLD);
  iguales = 0;
  for (i=0;i<np;i++) {
    if (vnum[i]==num) iguales++;
  }
} while (iguales!=np && cont<1000);

if (!p) {
  if (iguales==np)
    printf("Han pensado el mismo número en la vez %d\n",cont);
  else
    printf("ERR: Tras 1000 veces no coinciden los números\n");
}
free(vnum);
```

Cuestión 2–5

Se pretende implementar un generador de números aleatorios paralelo. Dados p procesos MPI, el programa funcionará de la siguiente forma: todos los procesos van generando una secuencia de números hasta que P_0 les indica que paren. En ese momento, cada proceso enviará a P_0 su último número generado y P_0 combinará todos esos números con el número que ha generado él. En pseudocódigo sería algo así:

```
n = inicial(id)
si id=0
  para i=1 hasta 100
    n = siguiente(n)
  fpara
  envia mensaje de aviso a procesos 1..np-1
  recibe m[k] de proceso k para k=1..np-1
  n = combina(n,m[k]) para k=1..np-1
si no
  n = inicial
  mientras no recibo mensaje de 0
    n = siguiente(n)
  fmientras
  envía n a 0
fsi
```

Implementar en MPI un esquema de comunicación asíncrona para este algoritmo, utilizando `MPI_Irecv` y `MPI_Test`. La recogida de resultados puede realizarse con una operación colectiva.

Solución: En el mensaje de aviso no es necesario enviar ningún dato, por lo que el buffer será un puntero nulo y la longitud 0.

```
MPI_Comm_rank(MPI_COMM_WORLD,&id);
MPI_Comm_size(MPI_COMM_WORLD,&np);
n = inicial(id);
```

```
if (id==0) {
  for (i=1;i<=100;i++) n = siguiente(n);
  for (k=1;k<np;k++) {
    MPI_Send(NULL,0,MPI_INT,k,1,MPI_COMM_WORLD);
  }
} else {
  MPI_Irecv(NULL,0,MPI_INT,0,1,MPI_COMM_WORLD,&req);
  do {
    n = siguiente(n);
    MPI_Test(&req,&flag,MPI_STATUS_IGNORE);
  } while (!flag);
}
MPI_Gather(&n,1,MPI_DOUBLE,m,1,MPI_DOUBLE,0,MPI_COMM_WORLD);
if (id==0) {
  for (k=1;k<np;k++) n = combina(n,m[k]);
}
```

Cuestión 2–6

Dado el siguiente fragmento de programa que calcula un valor aproximado para el número π:

```
double rx, ry, computed_pi;
long int i, points=NPOINTS, hits;
unsigned int seed = 1234;

hits = 0;
for (i=0; i<points; i++) {
  rx = (double)rand_r(&seed)/RAND_MAX;
  ry = (double)rand_r(&seed)/RAND_MAX;
  if ((rx-0.5)*(rx-0.5)+(ry-0.5)*(ry-0.5)<0.25) hits++;
}
computed_pi = 4.0*hits/points;
printf("Computed PI = %.10f\n", computed_pi);
```

Implementar una versión en MPI que permita su cálculo en paralelo.

Solución: La paralelización es sencilla en este caso dado que el programa es muy paralelizable. Consiste en la utilización correcta de la rutina `MPI_Reduce`. Cada proceso sólo tiene que calcular la cantidad de números aleatorios que debe generar y generarlos contabilizando los que caen dentro del círculo. Se multiplica el valor de la semilla por el identificador del proceso para que la secuencia de números aleatoria sea diferente en cada proceso.

```
double rx, ry, computed_pi;
long int i, points_per_proc, points=NPOINTS, hitproc, hits;
int myproc, nprocs;

MPI_Comm_rank(MPI_COMM_WORLD, &myproc);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

seed = myproc*1234;
points_per_proc = points/nprocs;
hitproc = 0;
for (i=0; i<points_per_proc; i++) {
```

```
    rx = (double)rand_r(&seed)/RAND_MAX;
    ry = (double)rand_r(&seed)/RAND_MAX;
    if ((rx-0.5)*(rx-0.5)+(ry-0.5)*(ry-0.5)<0.25) hitproc++;
  }

  MPI_Reduce(&hitproc,&hits,1,MPI_LONG,MPI_SUM,0,MPI_COMM_WORLD);

  if (!myproc) {
    computed_pi = 4.0*hits/points;
    printf("Computed PI = %.10f\n", computed_pi);
  }
```

Cuestión 2–7

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;
}
```

(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])
```

Solución:

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

double infNormPar(double A[][N], double ALocal[][N]) {
  int i,j,p;
  double s,nrm,nl=0.0;
```

```
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    MPI_Scatter(A,N*N/p,MPI_DOUBLE,ALocal,N*N/p,MPI_DOUBLE,0,
                MPI_COMM_WORLD);
    for (i=0; i<N/p; i++) {
      s=0.0;
      for (j=0; j<N; j++)
        s+=fabs(Alocal[i][j]);
      if (s>nl)
        nl=s;
    }
    MPI_Reduce(&nl,&nrm,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
    return nrm;
  }
```

(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.

Solución: Denotamos por n el tamaño del problema (`N`). El coste incluye tres etapas:

- Coste del reparto: $(p-1)\cdot\left(t_s + n\cdot\frac{n}{p}\cdot t_w\right)$
- Coste del procesado de $\frac{n}{p}$ filas: $\frac{n}{p}\cdot n = \frac{n^2}{p}$
- Coste de la reducción (implementación trivial): $(p-1)\cdot(t_s+t_w)$

Por tanto, el coste total es aproximadamente: $2\cdot p\cdot t_s + n^2\cdot t_w + \frac{n^2}{p}$

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

Solución: El coste computacional de la versión secuencial es aproximadamente n^2. Por tanto, el speed-up es $S(n,p) = \frac{n^2}{2\cdot p\cdot t_s + n^2\cdot t_w + \frac{n^2}{p}}$ y la eficiencia $E(n,p) = \frac{n^2}{2\cdot p^2\cdot t_s + p\cdot n^2\cdot t_w + n^2}$.
Los valores asintóticos del speed-up y la eficiencia cuando el tamaño del problema tiende a infinito son los siguientes:

$$\lim_{n\to\infty} S(n,p) = \lim_{n\to\infty} \frac{1}{2\cdot\frac{p}{n^2}\cdot t_s + t_w + \frac{1}{p}} = \frac{p}{p\cdot t_w + 1}$$

$$\lim_{n\to\infty} E(n,p) = \lim_{n\to\infty} \frac{1}{2\cdot\frac{p^2}{n^2}\cdot t_s + p\cdot t_w + 1} = \frac{1}{p\cdot t_w + 1}$$

Cuestión 2–8

Sea el siguiente código:

```
for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    w[j] = procesar(j, n, v);
  }
  for (j=0; j<n; j++) {
    v[j] = w[j];
  }
}
```

donde la función `procesar` tiene la siguiente cabecera:

```
double procesar(int j, int n, double *v);
```

siendo todos los argumentos solo de entrada.

(a) Indica su coste teórico (en flops) suponiendo que el coste de la función `procesar` es $2n$ flops.

Solución: Coste secuencial: $2n^2m$ flops.

(b) Paraleliza dicho código en MPI y justifícalo. Se supone que n es la dimensión de los vectores `v` y `w`, pero también el número de procesos MPI. La variable `p` contiene el identificador de proceso. El proceso `p=0` es el único que tiene el valor inicial del vector `v`. Se valora la manera más eficiente de realizar esta paralelización. Esto consiste en utilizar la/s rutina/s MPI adecuadas de manera que el número de las mismas sea mínimo.

Solución: En primer lugar, el proceso 0 difunde el vector a los demás procesos dado que la función `procesar` necesita este vector. El bucle externo no se puede paralelizar. Paralelizamos, por tanto, los bucles internos. En una iteración (`i`), el proceso `p` se encargará de ejecutar la función `procesar` y almacenar el resultado en la variable `a`. La actualización del vector `v` en el segundo bucle corresponde a un reparto en el que cada proceso envía a todos los demás el dato calculado en la variable `a`.

```
MPI_Bcast(v, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
for (i=0; i<m; i++) {
  double a = procesar(p, n, v);
  MPI_Allgather(&a, 1, MPI_DOUBLE, v, 1, MPI_DOUBLE, MPI_COMM_WORLD);
}
```

(c) Indica el coste de comunicaciones suponiendo que los nodos están conectados en una topología de tipo bus.

Solución: El coste de una difusión de n elementos en un bus es de $\beta + n\tau$. El coste de la operación de recogida de un elemento de cada proceso por parte de todos los procesos es $n(\beta + \tau)$. El coste total, teniendo en cuenta el número de iteraciones del bucle exterior, es de

$$(\beta + n\tau) + mn(\beta + \tau) \ .$$

(d) Indica la eficiencia alcanzable teniendo en cuenta que tanto m como n son grandes.

Solución: El speedup S_n se calcula como la ratio entre el tiempo secuencial y el paralelo (suponiendo n procesos):

$$S_n = \frac{2mn^2}{2mn + mn(\beta + \tau)} = \frac{2n}{2 + \beta + \tau} \ ,$$

donde se ha tenido en cuenta que el coste de la difusión es despreciable (frente a valores de m y n grandes). Por lo tanto, la eficiencia E para n procesos sería

$$E_n = \frac{S_n}{n} = \frac{2}{2 + \beta + \tau} \ .$$

Cuestión 2–9

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;
}
```

(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.

Solución: La matriz `A` se distribuye por bloques de filas consecutivas entre los procesos.

```
...
int main(int argc, char *argv[])
{
  double A[DIM][DIM], Aloc[DIM][DIM], x;
  int i, j, cont, cont_loc;
  int p, rank;

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

  /* Distribuir datos */
  MPI_Bcast(&x, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Scatter(A, DIM/p*DIM, MPI_DOUBLE, Aloc, DIM/p*DIM,
              MPI_DOUBLE, 0, MPI_COMM_WORLD);

  /* Calculo local */
  cont_loc=0;
  for (i=0; i<DIM/p; i++)
    for (j=0; j<DIM; j++)
      if (Aloc[i][j]==x) cont_loc++;

  /* Recoger resultado global en proc 0 */
  MPI_Reduce(&cont_loc, &cont, 1, MPI_INT, MPI_SUM, 0,
             MPI_COMM_WORLD);
  if (rank==0) printf("%d ocurrencias\n", cont);

  MPI_Finalize();
  return 0;
}
```

(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.

Solución: Por comodidad, llamamos n a la dimensión de la matriz (`DIM`).
El coste paralelo será la suma del coste aritmético más el coste de comunicaciones. El primero es:

$$t_a(n,p) = \sum_{i=0}^{n/p-1} \sum_{j=0}^{n-1} 1 = n^2/p$$

Respecto a las comunicaciones, supondremos que la difusión o *broadcast* realiza el envío de $p-1$ mensajes (desde el proceso raíz a cada uno de los restantes), y lo mismo ocurre con el *scatter* y con la reducción. Por tanto, el coste de comunicaciones será:

$$t_c(n,p) = 2(p-1)(t_s + t_w) + (p-1)(t_s + \frac{n^2}{p} t_w) \approx 3pt_s + (n^2 + 2p)t_w$$

Con lo cual, el coste paralelo será:

$$t(n,p) \approx n^2/p + 3pt_s + (n^2 + 2p)t_w$$

Cuestión 2–10

(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])
```

Solución:

```
void suma_mat(double a[N][N],double b[N][N])
{
   int i,j,np,tb;
   double al[N][N],bl[N][N];
   MPI_Comm_size(MPI_COMM_WORLD, &np);
   tb=N/np;
   MPI_Scatter(a, tb*N, MPI_DOUBLE, al, tb*N, MPI_DOUBLE, 0,
               MPI_COMM_WORLD);
   MPI_Scatter(b, tb*N, MPI_DOUBLE, bl, tb*N, MPI_DOUBLE, 0,
               MPI_COMM_WORLD);
   for(i=0;i<tb;i++)
      for(j=0;j<N;j++)
         al[i][j]+=bl[i][j];
   MPI_Gather(al, tb*N, MPI_DOUBLE, a, tb*N, MPI_DOUBLE, 0,
              MPI_COMM_WORLD);
}
```

(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.

Solución:

Tiempo secuencial: N^2 flops.

Tiempo paralelo, suponiendo que tenemos p procesos:

El coste de repartir una matriz cuadrada de orden N con `MPI_Scatter` (suponiendo un algoritmo trivial) se corresponde al coste del envío de $p-1$ mensajes de tamaño igual al número de elementos de cada bloque, es decir $\frac{N}{p}N = \frac{N^2}{p}$ elementos. Por tanto, el coste de distribuir las matrices `a` y `b` será

$$2(p-1)\left(t_s + \frac{N^2}{p}t_w\right) \approx 2pt_s + 2N^2t_w$$

donde para hacer la simplificación hemos supuesto un valor de p grande.

El coste paralelo de calcular concurrentemente la suma de las porciones locales de las matrices `a` y `b` es

$$\sum_{i=0}^{\frac{N}{p}-1}\sum_{j=0}^{N-1} = \sum_{i=0}^{\frac{N}{p}-1} N = \frac{N^2}{p} \quad \text{flops.}$$

El coste de que P_0 recoja en `a` el resultado de la suma (`MPI_Gather`) se corresponde al envío de un mensaje de cada proceso P_i $(i > 0)$ al proceso P_0 de tamaño igual a $\frac{N}{p}N = \frac{N^2}{p}$ elementos. Por tanto, el coste será:

$$(p-1)\left(t_s + \frac{N^2}{p}t_w\right) \approx pt_s + N^2t_w$$

Sumando los tres tiempos anteriores, tenemos que el coste paralelo es:

$$3pt_s + 3N^2t_w + \frac{N^2}{p}$$

Speedup:

$$S(N,p) = \frac{N^2}{3pt_s + 3N^2t_w + \frac{N^2}{p}}$$

Eficiencia:

$$E(N,p) = \frac{S(N,p)}{p} = \frac{N^2}{3p^2t_s + 3pN^2t_w + N^2}$$

Cuestión 2–11

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;
}
```

(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];
```

Solución:

```
double pscalarprod(double X[], double Y[], int n)
{
  double Xlcl[MAX], Ylcl[MAX];
  double prod=0.0, prodf;
  int i, p, nb;

  MPI_Comm_size(MPI_COMM_WORLD, &p);
  nb = n/p;
  MPI_Scatter(X, nb, MPI_DOUBLE, Xlcl, nb, MPI_DOUBLE, 0,
              MPI_COMM_WORLD);
  MPI_Scatter(Y, nb, MPI_DOUBLE, Ylcl, nb, MPI_DOUBLE, 0,
              MPI_COMM_WORLD);
  for (i=0;i<nb;i++)
    prod += Xlcl[i]*Ylcl[i];
  MPI_Reduce(&prod, &prodf, 1, MPI_DOUBLE, MPI_SUM, 0,
             MPI_COMM_WORLD);
  return prodf;
}
```

(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?

Solución: $t(n) = \sum_{i=0}^{n-1} 2 = 2n$ Flops

$$t(n,p) = 2(p-1)(t_s + \frac{n}{p}t_w) + \frac{2n}{p} + (p-1)(t_s + t_w) + p - 1 \approx 3pt_s + (2n+p)t_w + \frac{2n}{p}$$

$$S(n,p) = \frac{t(n)}{t(n,p)} = \frac{2n}{3p \cdot t_s + (2n+p)t_w + \frac{2n}{p}}$$

$$\lim_{n\to\infty} S(n,p) = \frac{2}{2 \cdot t_w + \frac{2}{p}} = \frac{2p}{2p \cdot t_w + 2}$$

Si $t_w = 0.1$ Flops, entonces el $S(n,p)$ estaría limitado por $\frac{2p}{0,2p} = 10$.

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

Solución: La llamada a `MPI_Reduce` se cambia por la siguiente línea:

```
MPI_Allreduce(&prod, &prodf, 1, MPI_DOUBLE, MPI_SUM,
              MPI_COMM_WORLD);
```

Cuestión 2–12

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];
```

(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 */
```

Solución:

```
int i, j, rank, p, bs;
double A[N][N], B[N][N];

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);
if (rank==0)
    leer(A);
bs = N/p;
MPI_Scatter(A,bs*N,MPI_DOUBLE,B,bs*N,MPI_DOUBLE,0,MPI_COMM_WORLD);
for (i=0;i<bs;i++)
    for(j=0;j<N;j++)
        B[i][j]= B[i][j]*B[i][j];
MPI_Gather(B,bs*N,MPI_DOUBLE,A,bs*N,MPI_DOUBLE,0,MPI_COMM_WORLD);
```

(b) Calcula el speedup y la eficiencia.

Solución: El coste computacional secuencial es $t(N) = N^2$ flops.
Como el reparto (`MPI_Scatter`) o recogida (`MPI_Gather`) de una matriz de orden N entre p procesos comporta el envío/recepción de $p-1$ mensajes de tamaño $\frac{N^2}{p}$, el tiempo de comunicaciones es $t_c = 2(p-1)\left(t_s + \frac{N^2}{p}t_w\right)$. El coste aritmético paralelo es $\frac{N^2}{p}$. Por lo tanto, el tiempo total paralelo resulta ser:

$$t(N,p) = 2(p-1)\left(t_s + \frac{N^2}{p}t_w\right) + \frac{N^2}{p}.$$

Luego el speedup es igual a

$$S(N,p) = \frac{t(N)}{t(N,p)} = \frac{N^2}{2(p-1)\left(t_s + \frac{N^2}{p}t_w\right) + \frac{N^2}{p}}$$

y la eficiencia

$$E(N,p) = \frac{S(N,p)}{p} = \frac{N^2}{2p(p-1)\left(t_s + \frac{N^2}{p}t_w\right) + N^2}$$

Cuestión 2–13

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;
}
```

(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.

Solución:

```
int main(int argc, char *argv[])
{
  int i,j,id,p,tb;
  double A[N][N],Al[N][N],v[N],vl[N];
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  if (id==0) read_mat(A);
  tb = N/p;
  MPI_Scatter(A, tb*N, MPI_DOUBLE, Al, tb*N, MPI_DOUBLE, 0,
              MPI_COMM_WORLD);
  for (i=0;i<tb;i++) {
    vl[i] = 0.0;
    for (j=0;j<N;j++)
      vl[i] += Al[i][j];
  }
  MPI_Gather(vl, tb, MPI_DOUBLE, v, tb, MPI_DOUBLE, 0,
             MPI_COMM_WORLD);
  if (id==0) write_vec(v);
  MPI_Finalize();
  return 0;
}
```

(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.

Solución: Tiempo secuencial:

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

Tiempo paralelo aritmético con p procesos:

$$t_{arit}(N,p) = \sum_{i=0}^{N/p-1} \sum_{j=0}^{N-1} 1 = \frac{N^2}{p} \text{ flops}$$

Tiempo paralelo de comunicaciones con p procesos:

- Reparto de la matriz A:
$$(p-1)\left(t_s + \frac{N^2}{p} t_w\right)$$
- Recogida del vector v:
$$(p-1)\left(t_s + \frac{N}{p} t_w\right)$$

Por lo tanto, el tiempo paralelo es

$$t(N,p) = \frac{N^2}{p} \text{ flops} + (p-1)\left(2t_s + \frac{N}{p}(N+1)t_w\right) \approx \frac{N^2}{p} \text{ flops} + 2pt_s + N^2 t_w$$

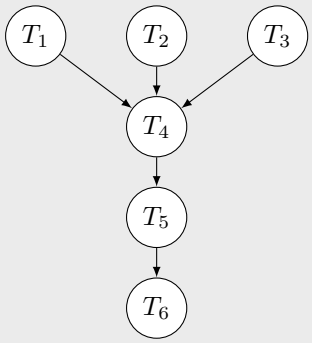
Cuestión 2–14

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;
}
```

(a) Dibuja el grafo de dependencias y calcula el coste secuencial.

Solución: El grafo de dependencias se muestra en la siguiente figura:



El coste secuencial es: $t(n) = 7n + 7n + 7n + 2 + n + n \approx 23n$

(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.

Solución: Las tres primeras tareas se deben asignar a procesos distintos, para repartir la carga. Las otras tres tareas se han de ejecutar en el orden secuencial (debido a las dependencias), por lo que las asignamos a P_0, ya que este proceso es el que debe almacenar el valor correcto de retorno de la función.

```
double ejemplo(int val[3])
{
  double a,b,c,d,e,f,operand;
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  switch (rank) {
    case 0: a = T1(val[0]); operand = a; break;
    case 1: b = T2(val[1]); operand = b; break;
    case 2: c = T3(val[2]); operand = c; break;
  }
  MPI_Reduce(&operand, &d, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
  if (rank==0) {
    e = T5(val[2],d);
    f = T6(val[0],val[1],e);
  }
  return f;
}
```

(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.

Solución: El tiempo paralelo se calcula a partir del coste asociado al camino crítico del grafo de dependencias, correspondiente a $T_1 - T_4 - T_5 - T_6$, por ejemplo. Para el coste de comunicación, consideramos que la operación de reducción internamente enviará únicamente dos mensajes, cada uno de ellos de longitud igual a 1 `double` (en la reducción también se realizarán 2 flops, que no se han incluido en las expresiones).

$$t(n,3) = t_{arit}(n,3) + t_{comm}(n,3)$$
$$t_{arit}(n,3) = 7n + n + n \approx 9n$$

$$t_{comm}(n,3) = 2 \cdot (t_s + t_w)$$

$$t(n,3) \approx 9n + 2t_s + 2t_w$$

El speedup será:

$$S(n,3) = \frac{t(n)}{t(n,3)} \approx \frac{23n}{9n + 2t_s + 2t_w}$$

Como en este caso el coste de comunicación es muy bajo, asintóticamente (cuando n crece) el speedup tiende al valor $S(n,3) \to \frac{23n}{9n} = 2,56$.

Cuestión 2–15

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;
}
```

(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.

Solución:

```
int cuenta(double v[], int n)
{
  int i, cont, cont_loc=0, p;
  double media, suma_loc=0;
  double *vloc;

  MPI_Comm_size(MPI_COMM_WORLD, &p);
  vloc = (double*) malloc(n/p*sizeof(double));

  MPI_Scatter(v, n/p, MPI_DOUBLE, vloc, n/p, MPI_DOUBLE, 0,
              MPI_COMM_WORLD);

  for (i=0;i<n/p;i++)
    suma_loc += vloc[i];

  MPI_Allreduce(&suma_loc, &media, 1, MPI_DOUBLE, MPI_SUM,
                MPI_COMM_WORLD);
```

```
    media = media/n;

    for (i=0;i<n/p;i++)
      if (vloc[i]>media/2 && vloc[i]<media*2)
        cont_loc++;

    MPI_Reduce(&cont_loc, &cont, 1, MPI_INT, MPI_SUM, 0,
               MPI_COMM_WORLD);
    free(vloc);
    return cont;
  }
```

(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.

Solución:

- Scatter: el proceso 0 envía un mensaje de n/p elementos a cada uno de los demás procesos. Por tanto:
$$(p-1)(t_s + \frac{n}{p}t_w)$$
- Reduce: el proceso 0 recibe un mensaje de un elemento de cada uno de los demás procesos, y suma los elementos, es decir:
$$(p-1)(t_s + t_w) + (p-1)$$
- Allreduce: se puede realizar mediante una operación *reduce* sobre el proceso 0, seguida de un *broadcast* del resultado. Para el broadcast, suponemos que el proceso 0 envía un mensaje de un elemento a cada uno de los demás procesos.
$$2(p-1)(t_s + t_w) + (p-1)$$
- Bucles de cálculo:
$$\sum_{i=0}^{\frac{n}{p}-1} 1 + \sum_{i=0}^{\frac{n}{p}-1} 2 = \frac{3n}{p}$$

El tiempo de ejecución paralelo es la suma de lo anterior, es decir:

$$t(n,p) \approx 4pt_s + (n+3p)t_w + 2p + \frac{3n}{p}$$

Por otra parte, el tiempo secuencial es:

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

De donde tenemos:

$$S(n,p) = \frac{3n}{4pt_s + (n+3p)t_w + 2p + \frac{3n}{p}}$$

$$\lim_{n\to\infty} S(n,p) = \frac{3}{t_w + 3/p}$$

Cuestión 2–16

El siguiente programa secuencial realiza ciertos cálculos sobre una matriz cuadrada `A`.

```
#define N ...
int i, j;
double A[N][N], sum[N], fact, max;
...
for (i=0;i<N;i++) {
  sum[i] = 0.0;
  for (j=0;j<N;j++) sum[i] += A[i][j]*A[i][j];
}
fact = 1.0/sum[0];
for (i=0;i<N;i++) sum[i] *= fact;

max = 0.0;
for (i=0;i<N;i++) {
  if (sum[i]>max) max = sum[i];
}
for (i=0;i<N;i++) {
  for (j=0;j<N;j++) A[i][j] *= max;
}
```

(a) Paraleliza el código mediante MPI suponiendo que cada proceso tiene ya almacenadas `k=N/p` filas consecutivas de la matriz, siendo `p` el número de procesos (se puede asumir que `N` es divisible entre `p`). Estas filas ocupan las primeras posiciones de la matriz local, es decir, entre las filas `0` y `k-1` de la variable `A`. Nota: Deben utilizarse primitivas de comunicación colectiva siempre que sea posible.

Solución: Cada proceso calcula una parte del vector de sumas `sum`, almacenando los valores en las primeras `k` posiciones. El valor de `fact` se calcula en P_0 (que es quien tiene la fila 0 de la matriz) y se difunde al resto de procesos. En la última parte del algoritmo, cada proceso calcula el máximo local y se realiza una reducción cuyo resultado (`max`) es necesario tenerlo en todos los procesos.

```
#define N ...
int i, j, k, rank, p;
double A[N][N], sum[N], fact, max, maxloc;
...
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);
k = N/p;
for (i=0;i<k;i++) {
  sum[i] = 0.0;
  for (j=0;j<N;j++) sum[i] += A[i][j]*A[i][j];
}
if (rank==0) fact = 1.0/sum[0];
MPI_Bcast(&fact, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
for (i=0;i<k;i++) sum[i] *= fact;

maxloc = 0.0;
for (i=0;i<k;i++) {
  if (sum[i]>maxloc) maxloc = sum[i];
}
MPI_Allreduce(&maxloc, &max, 1, MPI_DOUBLE, MPI_MAX,
              MPI_COMM_WORLD);
```

```
    for (i=0;i<k;i++) {
      for (j=0;j<N;j++) A[i][j] *= max;
    }
```

(b) Escribe el código para realizar la comunicación necesaria tras el cálculo anterior para que la matriz completa quede almacenada en el proceso 0 en la variable `Aglobal[N][N]`.

Solución:

```
    MPI_Gather(A, k*N, MPI_DOUBLE, Aglobal, k*N, MPI_DOUBLE, 0,
               MPI_COMM_WORLD);
```

Cuestión 2–17

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.

Solución:

```
int main(int argc, char *argv[]) {
  int i, j, k, p, myid;
  double a, b, A[M][N], B[M][N], C[M][N];
  double Alocal[M][N], Blocal[M][N], Clocal[M][N];
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD ,&p);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  if (myid==0) LeeOperandos(A, B, &a, &b);
  MPI_Bcast(&a, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast(&b, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  k=M/p;
```

```
  for (i=0; i<k; i++) {
    MPI_Scatter(&A[i*p][0], N, MPI_DOUBLE, &Alocal[i][0], N,
                MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Scatter(&B[i*p][0], N, MPI_DOUBLE, &Blocal[i][0], N,
                MPI_DOUBLE, 0, MPI_COMM_WORLD);
  }
  for (i=0; i<k; i++) {
    for (j=0; j<N; j++) {
      Clocal[i][j] = a*Alocal[i][j] + b*Blocal[i][j];
    }
  }
  for (i=0; i<k; i++) {
    MPI_Gather(&Clocal[i][0], N, MPI_DOUBLE, &C[i*p][0], N,
               MPI_DOUBLE, 0, MPI_COMM_WORLD);
  }
  if (myid==0) EscribeMatriz(C);
  MPI_Finalize();
  return 0;
}
```

Cuestión 2–18

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.

Solución:

```
void funcpar(double A[M][N], int sup[M]) {
   double Aloc[M][N];
   int suploc[M];
   double sumaloc, media;
```

```
        int i, j, p;
        MPI_Comm_size(MPI_COMM_WORLD, &p);
        MPI_Scatter(A, M*N/p, MPI_DOUBLE, Aloc, M*N/p, MPI_DOUBLE, 0,
                    MPI_COMM_WORLD);
        sumaloc=0;
        for (i=0; i<M/p; i++)
           for (j=0; j<N; j++)
              sumaloc += Aloc[i][j];
        MPI_Allreduce(&sumaloc, &media, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
        media = media/(M*N);
        for (i=0; i<M/p; i++) {
           suploc[i] = 0;
           for (j=0; j<N; j++)
              if (Aloc[i][j]>media) suploc[i]++;
        }
        MPI_Gather(suploc, M/p, MPI_INT, sup, M/p, MPI_INT, 0, MPI_COMM_WORLD);
     }
```

Cuestión 2–19

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;
}
```

(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`.

Solución: La función `facto` no requiere modificación. En el programa principal, hacemos una distribución por bloques del vector.

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

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&id);
  MPI_Comm_size(MPI_COMM_WORLD,&np);
```

```
  if (id==0) n = lee_vector(v);
  MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
  k = n / np;
  MPI_Scatter(v,k,MPI_DOUBLE,vloc,k,MPI_DOUBLE,0,MPI_COMM_WORLD);
  for (i=0; i<k; i++) {
    vloc[i] = facto(n,vloc[i]);
    a = a * vloc[i];
  }
  MPI_Gather(vloc,k,MPI_DOUBLE,v,k,MPI_DOUBLE,0,MPI_COMM_WORLD);
  MPI_Reduce(&a,&total,1,MPI_DOUBLE,MPI_PROD,0,MPI_COMM_WORLD);
  if (id==0) {
    printf("Factor alfalfa: %.2f\n",total);
    escribe_vector(n,v);
  }
  MPI_Finalize();
  return 0;
}
```

(b) Calcula el tiempo de ejecución secuencial.

Solución:

$$t_{\text{facto}}(m) = \sum_{i=1}^{m} 2 = 2m \text{ flops}$$

$$t_1(n) = \sum_{i=0}^{n-1} (1 + t_{\text{facto}}(n)) = \sum_{i=0}^{n-1} (1 + 2n) = n + 2n^2 \approx 2n^2 \text{ flops}$$

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

Solución:

$$t_p(n) = t_{\text{Bcast}} + t_{\text{Scatter}} + \frac{n + 2n^2}{p} \text{ flops} + t_{\text{Gather}} + t_{\text{Reduce}}$$

$$t_{\text{Bcast}} = (p-1)(t_s + t_w)$$

$$t_{\text{Scatter}} = t_{\text{Gather}} = (p-1)\left(t_s + \frac{n}{p} t_w\right)$$

$$t_{\text{Reduce}} = (p-1)(t_s + t_w + 1 \text{ flops})$$

Cuestión 2–20

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;
}
```

(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.

Solución:

```
double suma(double v[N])
{
  int i, id, np, nb, p;
  double s, sl = 0.0, vl[N];
  MPI_Comm_rank(MPI_COMM_WORLD,&id);
  MPI_Comm_size(MPI_COMM_WORLD,&np);
  nb = N / np;
  if (id==0) {
    for (p=1; p<np; p++)
      MPI_Send(&v[p*nb],nb,MPI_DOUBLE,p,22,MPI_COMM_WORLD);
    for (i=0; i<nb; i++) sl += v[i];
    s = sl;
    for (p=1; p<np; p++) {
      MPI_Recv(&sl,1,MPI_DOUBLE,p,23,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
      s += sl;
    }
    for (p=1; p<np; p++)
      MPI_Send(&s,1,MPI_DOUBLE,p,24,MPI_COMM_WORLD);
  } else {
    MPI_Recv(vl,nb,MPI_DOUBLE,0,22,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    for (i=0; i<nb; i++) sl += vl[i];
    MPI_Send(&sl,1,MPI_DOUBLE,0,23,MPI_COMM_WORLD);
    MPI_Recv(&s,1,MPI_DOUBLE,0,24,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
  }
  return s;
}
```

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

Solución:

```
double suma(double v[N])
{
  int i,np,nb;
  double s, sl = 0, vl[N];
  MPI_Comm_size(MPI_COMM_WORLD,&np);
  nb = N / np;
  MPI_Scatter(v,nb,MPI_DOUBLE,vl,nb,MPI_DOUBLE,0,MPI_COMM_WORLD);
  for (i=0; i<nb; i++) sl += vl[i];
  MPI_Allreduce(&sl,&s,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
  return s;
}
```

Cuestión 2–21

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);
}
```

(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];
```

Solución: Podemos usar una distribución por bloques de filas de la matriz:

```
void par_search(double A[M][N], double x, int owner) {
  double Aloc[M][N];
  int i,j,first,count, fl,cl, id,np, rows,size;

  MPI_Comm_rank(MPI_COMM_WORLD,&id);
  MPI_Comm_size(MPI_COMM_WORLD,&np);
  rows=M/np; size=rows*N;

  /* Reparto de la matriz por bloques de filas */
  MPI_Scatter(A,size,MPI_DOUBLE,Aloc,size,MPI_DOUBLE,owner,MPI_COMM_WORLD);
  /* Difusión del número a buscar */
  MPI_Bcast(&x,1,MPI_DOUBLE,owner,MPI_COMM_WORLD);

  /* Cálculos locales */
  fl = M ; cl = 0;
  for (i=0; i<rows; i++)
    for (j=0; j<N; j++)
      if (Aloc[i][j] == x) {
        cl++;
        if (i < fl) fl = i;
      }

  /* Recogida de resultados en un proceso e impresión */
  fl = rows*id + fl;  /* Pasar índice local a global */
  MPI_Reduce(&fl,&first,1,MPI_INT,MPI_MIN,0,MPI_COMM_WORLD);
  MPI_Reduce(&cl,&count,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
  if (id == 0)
    printf("%g está %d veces, la primera vez en la fila %d.\n",x,count,first);
}
```

(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.

Solución:

$$t_{\text{scatter}} = (p-1)\cdot(t_s + \frac{MN}{p}t_w)$$
$$t_{\text{bcast}} = (p-1)\cdot(t_s + t_w)$$
$$t_{\text{reduce}} = (p-1)\cdot(t_s + t_w)$$

Cuestión 2–22

(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.

Solución:

```
MPI_Reduce(&val, &res, 1, MPI_DOUBLE, MPI_SUM, 0, comm);
```

(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.

Solución: Una posible implementación sería aquella en que el proceso 0 realiza los envíos mediante un bucle sencillo.

```
#define TAG 999
double val=...;
int i, sz, rank;
```

```
MPI_Status stat;
...
MPI_Comm_size(comm, &sz);
MPI_Comm_rank(comm, &rank);
if (rank==0) {
  for (i=1; i<sz; i++) {
    MPI_Send(&val, 1, MPI_DOUBLE, i, TAG, comm);
  }
} else {
  MPI_Recv(&val, 1, MPI_DOUBLE, 0, TAG, comm, &stat);
}
```

3. Tipos de datos

Cuestión 3–1

Supongamos definida una matriz de enteros `A[M][N]`. Escribe el fragmento de código necesario para el envío desde P_0 y la recepción en P_1 de los datos que se especifican en cada caso, usando para ello un solo mensaje. En caso necesario, define un tipo de datos derivado MPI.

(a) Envío de la tercera fila de la matriz `A`.

Solución: En el lenguaje C, los arrays bidimensionales se almacenan por filas, con lo que la separación entre elementos de la misma fila es 1. Por tanto, en este caso no es necesario crear un tipo MPI, por estar los elementos contiguos en memoria.

```
int A[M][N];
MPI_Status st;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank==0) {
  MPI_Send(&A[2][0], N, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (rank==1) {
  MPI_Recv(&A[2][0], N, MPI_INT, 0, 0, MPI_COMM_WORLD,&st);
}
```

(b) Envío de la tercera columna de la matriz `A`.

Solución: La separación entre elementos de la misma columna es `N`.

```
int A[M][N];
MPI_Status st;
MPI_Datatype newtype;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Type_vector(M, 1, N, MPI_INT, &newtype);
MPI_Type_commit(&newtype);
if (rank==0) {
  MPI_Send(&A[0][2], 1, newtype, 1, 0, MPI_COMM_WORLD);
} else if (rank==1) {
  MPI_Recv(&A[0][2], 1, newtype, 0, 0, MPI_COMM_WORLD,&st);
}
MPI_Type_free(&newtype);
```

Cuestión 3–2

Dado el siguiente fragmento de un programa MPI:

```
struct Tdatos {
  int x;
  int y[N];
  double a[N];
};
void distribuye(struct Tdatos *datos, int n, MPI_Comm comm) {
  int p, pr, pr2;
  MPI_Status status;

  MPI_Comm_size(comm, &p);
  MPI_Comm_rank(comm, &pr);
  if (pr==0) {
    for (pr2=1; pr2<p; pr2++) {
      MPI_Send(&(datos->x), 1, MPI_INT, pr2, 0, comm);
      MPI_Send(&(datos->y[0]), n, MPI_INT, pr2, 0, comm);
      MPI_Send(&(datos->a[0]), n, MPI_DOUBLE, pr2, 0, comm);
    }
  } else {
    MPI_Recv(&(datos->x), 1, MPI_INT, 0, 0, comm, &status);
    MPI_Recv(&(datos->y[0]), n, MPI_INT, 0, 0, comm, &status);
    MPI_Recv(&(datos->a[0]), n, MPI_DOUBLE, 0, 0, comm, &status);
  }
}
```

Modificar la función `distribuye_datos` para optimizar las comunicaciones.

(a) Realiza una versión que utilice tipos de datos derivados de MPI, de forma que se realice un envío (a cada proceso) en lugar de tres.

Solución: Dado que los datos a enviar/recibir son de diferentes tipos, para poder enviarlos en un mismo mensaje se debe definir un tipo de datos mediante `MPI_Type_create_struct`.

```
void distribuye(struct Tdatos *datos, int n, MPI_Comm comm) {
  int p, pr, pr2;
  MPI_Status status;
  int longitudes[]={1,n,n};
  MPI_Datatype tipos[]={MPI_INT, MPI_INT, MPI_DOUBLE};
  MPI_Aint despls[3];
  MPI_Aint dir1, dirx, diry, dira;

  MPI_Comm_size(comm, &p);
  MPI_Comm_rank(comm, &pr);

  /* Calculo de los desplazamientos de cada componente */
  MPI_Get_address(datos, &dir1);
  MPI_Get_address(&(datos->x), &dirx);
  MPI_Get_address(&(datos->y[0]), &diry);
  MPI_Get_address(&(datos->a[0]), &dira);
  despls[0]=dirx-dir1;
  despls[1]=diry-dir1;
  despls[2]=dira-dir1;
```

```
  MPI_Type_create_struct(3,longitudes,despls,tipos,&Tnuevo);
  MPI_Type_commit(&Tnuevo);
  if (pr==0) {
    for (pr2=1; pr2<p; pr2++) {
      MPI_Send(datos, 1, Tnuevo, pr2, 0, comm);
    }
  }
  else {
    MPI_Recv(datos, 1, Tnuevo, 0, 0, comm, &status);
  }
  MPI_Type_free(&Tnuevo);
}
```

(b) Realiza una modificación de la anterior para que utilice primitivas de comunicación colectiva.

Solución: Sería idéntica a la anterior, excepto el último `if`, que se cambiaría por la siguiente instruccion:

```
MPI_Bcast(datos, 1, Tnuevo, 0, comm);
```

Cuestión 3–3

Se quiere implementar un programa paralelo para resolver el problema del Sudoku. Cada posible configuración del Sudoku o "tablero" se representa por una array de 81 enteros, conteniendo números entre 0 y 9 (0 representa una casilla vacía). El proceso 0 genera n soluciones, cuya validez deberá ser comprobada por los demás procesos. Estas soluciones se almacenan en una matriz A de tamaño $n \times 81$.

(a) Escribir el código que distribuye toda la matriz desde el proceso 0 hasta el resto de procesos de manera que cada proceso reciba un tablero (suponiendo $n = p$, donde p es el número de procesos).

Solución:

```
MPI_Scatter(A, 81, MPI_INT, tablero, 81, MPI_INT, 0, MPI_COMM_WORLD);
```

(b) Supongamos que para implementar el algoritmo en MPI creamos la siguiente estructura en C:

```
struct tarea {
  int tablero[81];
  int inicial[81];
  int es_solucion;
};
typedef struct tarea Tarea;
```

Crear un tipo de dato MPI `ttarea` que represente la estructura anterior.

Solución:

```
Tarea t;
MPI_Datatype ttarea;
int blocklen[3] = { 81, 81, 1 };
MPI_Aint ad1, ad2, ad3, ad4, disp[3];
MPI_Get_address(&t, &ad1);
MPI_Get_address(&t.tablero[0], &ad2);
MPI_Get_address(&t.inicial[0], &ad3);
MPI_Get_address(&t.es_solucion, &ad4);
disp[0] = ad2 - ad1;
disp[1] = ad3 - ad1;
```

```
disp[2] = ad4 - ad1;
MPI_Datatype types[3] = { MPI_INT, MPI_INT, MPI_INT };
MPI_Type_create_struct(3, blocklen, disp, types, &ttarea);
MPI_Type_commit(&ttarea);
```

Cuestión 3–4

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.

Solución: Puede verse como un vector de 3 bloques de elementos, cada uno de ellos con longitud 3 y con separación `N`.

```
double A[N][N];
int rank;
MPI_Datatype newtype;
... /* rellenar matriz */
MPI_Type_vector(3, 3, N, MPI_DOUBLE, &newtype);
MPI_Type_commit(&newtype);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank==0) {
  MPI_Send(&A[0][0], 1, newtype, 1, 0, MPI_COMM_WORLD);
} else if (rank==1) {
  MPI_Recv(&A[0][0], 1, newtype, 0, 0, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
}
MPI_Type_free(&newtype);
```

(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).

Solución: Bastaría con cambiar la dirección del buffer en `MPI_Send` y `MPI_Recv`. En concreto, habría que poner `&A[0][3]` en vez de `&A[0][0]` en la llamada a `MPI_Send`, y `&A[3][0]` en vez de `&A[0][0]` en la llamada a `MPI_Recv`.

Cuestión 3–5

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();
```

(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.

Solución:

Solución definiendo un nuevo tipo de dato:

```
MPI_Datatype cyclic_row;
mb = M/p;
MPI_Type_vector(mb, N, p*N, MPI_DOUBLE, &cyclic_row);
MPI_Type_commit(&cyclic_row);
if (rank==0) {
  for (i=1;i<p;i++) {
    MPI_Send(&A[i][0], 1, cyclic_row, i, 0, MPI_COMM_WORLD);
    MPI_Send(&B[i][0], 1, cyclic_row, i, 1, MPI_COMM_WORLD);
  }
  MPI_Sendrecv(A, 1, cyclic_row, 0, 0, Al, mb*N, MPI_DOUBLE,
               0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Sendrecv(B, 1, cyclic_row, 0, 1, Bl, mb*N, MPI_DOUBLE,
               0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
} else {
  MPI_Recv(Al, mb*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
  MPI_Recv(Bl, mb*N, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
}
```

La operación `MPI_Sendrecv` se ha usado para copiar la parte localmente almacenada en el proceso 0; también se podría hacer con un bucle o con `memcpy`.

Solución usando comunicaciones colectivas:

```
mb = M/p;
for (i=0;i<mb;i++) {
  MPI_Scatter(&A[i*p][0], N, MPI_DOUBLE, &Al[i][0], N,
              MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Scatter(&B[i*p][0], N, MPI_DOUBLE, &Bl[i][0], N,
              MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

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

Solución:

```
for (i=0;i<mb;i++)
```

```
      for (j=0;j<N;j++)
        Al[i][j] += Bl[i][j];
```

(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.

Solución: Solución usando el tipo de datos definido en el apartado (a):

```
if (rank==0) {
  for (i=1;i<p;i++)
    MPI_Recv(&A[i][0], 1, cyclic_row, i, 3, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
  MPI_Sendrecv(Al, mb*N, MPI_DOUBLE, 0, 3, A, 1, cyclic_row,
               0, 3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
else MPI_Send(Al, mb*N, MPI_DOUBLE, 0, 3, MPI_COMM_WORLD);
MPI_Type_free(&cyclic_row);
```

Solución usando comunicaciones colectivas:

```
for (i=0;i<mb;i++)
  MPI_Gather(&Al[i][0], N, MPI_DOUBLE, &A[i*p][0], N,
             MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Cuestión 3–6

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.

Solución:

```
void bcast_fila_col(double A[N][N], int k)
{
  MPI_Datatype colu;
  MPI_Type_vector(N, 1, N, MPI_DOUBLE, &colu);
  MPI_Type_commit(&colu);

  /* Envío de la fila */
  MPI_Bcast(&A[k][0], N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  /* Envío de la columna */
  MPI_Bcast(&A[0][k], 1, colu, 0, MPI_COMM_WORLD);

  MPI_Type_free(&colu);
}
```

Cuestión 3–7

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:

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

(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.

Solución:

```
void comunica(double A[2*N][2*N], double B[N][N])
{
  int rank;
  MPI_Datatype mat;

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Type_vector(N,N,2*N,MPI_DOUBLE,&mat);
  MPI_Type_commit(&mat);
  if (rank==0) {
    MPI_Sendrecv(&A[0][0],1,mat,0,0,B,N*N,MPI_DOUBLE,0,0,
                 MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    MPI_Send(&A[0][N],1,mat,1,0,MPI_COMM_WORLD);
    MPI_Send(&A[N][0],1,mat,2,0,MPI_COMM_WORLD);
    MPI_Send(&A[N][N],1,mat,3,0,MPI_COMM_WORLD);
  }
  else
    MPI_Recv(B,N*N,MPI_DOUBLE,0,0,MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
  MPI_Type_free(&mat);
}
```

(b) Calcula el tiempo de comunicaciones.

Solución: Como P_0 envía un total de tres mensajes de N^2 datos al resto de procesos, el tiempo de comunicaciones es $t_c = 3\left(t_s + N^2 t_w\right)$, siendo t_s el tiempo de establecimiento de la comunicación y t_w el tiempo necesario para enviar un dato.

Cuestión 3–8

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		

Solución:

```
void envia(int m,int n,double A[M][N],double v[MAX],MPI_Comm comm)
{
  int myid;
  MPI_Datatype mat;

  MPI_Comm_rank(comm,&myid);
  MPI_Type_vector(m,n,N,MPI_DOUBLE,&mat);
  MPI_Type_commit(&mat);
  if (myid == 0)
    MPI_Send(&A[0][0],1,mat,1,2512,comm);
  else if (myid == 1)
    MPI_Recv(&v[0],m*n,MPI_DOUBLE,0,2512,comm,MPI_STATUS_IGNORE);
  MPI_Type_free(&mat);
}
```

Cuestión 3–9

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

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

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]) {
```

Solución:

```
void SendBAD(double A[3*DIM][3*DIM], double Alcl[DIM][DIM]) {
  int i,rank,p,m,n;
  MPI_Datatype DiagBlock;

  MPI_Comm_size(MPI_COMM_WORLD, &p);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  n = 3*DIM;
  m = DIM;
  if (rank == 0) {
    MPI_Type_vector(m, m, n, MPI_DOUBLE, &DiagBlock);
    MPI_Type_commit(&DiagBlock);
    MPI_Sendrecv(A, 1, DiagBlock, 0, 0, Alcl, m*m, MPI_DOUBLE, 0,
                 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    for (i=1;i<p;i++)
      MPI_Send(&A[i*m][i*m], 1, DiagBlock, i, 0, MPI_COMM_WORLD);
    MPI_Type_free(&DiagBlock);
  } else {
    MPI_Recv(Alcl, m*m, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
  }
}
```

Cuestión 3–10

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])
```

Solución:

```
void distribuye(double A[NF][NC], double Aloc[NF][NC/2])
{
  int rank;
  MPI_Status stat;
  MPI_Datatype cols;
  MPI_Type_vector(NF, NC/2, NC, MPI_DOUBLE, &cols);
  MPI_Type_commit(&cols);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank==0) {
    MPI_Sendrecv(A, 1, cols, 0, 100, Aloc, NF*NC/2, MPI_DOUBLE,
                 0, 100, MPI_COMM_WORLD, &stat);
    MPI_Send(&A[0][NC/2], 1, cols, 1, 100, MPI_COMM_WORLD);
  }
  else if (rank==1) {
    MPI_Recv(Aloc,NF*NC/2,MPI_DOUBLE,0,100,MPI_COMM_WORLD,&stat);
```

```
    }
    MPI_Type_free(&cols);
  }
```

Cuestión 3–11

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

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

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

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

Solución:

```
void comunica(double A[N][N], double B[N/2][N])
{
  int id;
  MPI_Datatype mitad;
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Type_vector(N/2, N, 2*N, MPI_DOUBLE, &mitad);
  MPI_Type_commit(&mitad);
  if (id==0) {
    MPI_Send(&A[0][0], 1, mitad, 1, 100, MPI_COMM_WORLD);
    MPI_Send(&A[1][0], 1, mitad, 2, 100, MPI_COMM_WORLD);
  }
  else if (id==1 || id==2)
    MPI_Recv(B,N/2*N, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
  MPI_Type_free(&mitad);
}
```

(b) Calcula el tiempo de comunicaciones.

Solución: Se trata del envío/recepción de dos mensajes de $N^2/2$ datos; por lo tanto, el tiempo de comunicaciones es

$$t_c = 2\left(t_s + \frac{N^2}{2} t_w\right).$$

Cuestión 3–12

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).

Solución:

```
void envia_tridiagonal(double A[N][N],int root,MPI_Comm comm)
{
  MPI_Datatype diag3;
  MPI_Type_vector(N-2,3,N+1,MPI_DOUBLE,&diag3);
  MPI_Type_commit(&diag3);
  MPI_Bcast(&A[1][0],1,diag3,root,comm);
  MPI_Type_free(&diag3);
}
```

Cuestión 3–13

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);
}
```

(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.

Solución: Los procesos envían un solo mensaje de longitud `M*N` con la submatriz completa. Pero en el proceso P_0 estos datos no se almacenan contiguos en memoria, por lo que hay que definir un tipo "vector" de MPI. El tipo nuevo debe tener `M` bloques (uno por cada fila) de `N` elementos (tantos como columnas en el proceso que envía), con una separación entre los bloques (*stride*) de `N*p` ya que la matriz completa (en P_0) tiene `N*p` columnas. En la operación `MPI_Recv` el buffer debe apuntar a la primera posición de la submatriz correspondiente al proceso que envía (`k`), es decir, `aglobal[0][k*N]`.

```
int rank, i, j, k, p;
double alocal[M][N];
MPI_Datatype cols;
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 */
  MPI_Type_vector(M, N, N*p, MPI_DOUBLE, &cols);
  MPI_Type_commit(&cols);
  for (k=1;k<p;k++) {
    MPI_Recv(&aglobal[0][k*N], 1, cols, k, 33, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
  }
  MPI_Type_free(&cols);
  write(p,aglobal);
```

```
  } else {
    MPI_Send(&alocal[0][0], M*N, MPI_DOUBLE, 0, 33, MPI_COMM_WORLD);
  }
```

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

Solución: Versión original: se envían un total de $(p-1)M$ mensajes, cada uno de ellos de longitud N, por tanto

$$t_c = (p-1)M(t_s + Nt_w).$$

Versión modificada: se envían un total de $p-1$ mensajes, cada uno de ellos de longitud MN, por tanto

$$t_c = (p-1)(t_s + MNt_w).$$

Cuestión 3–14

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)
```

Solución:

```
void distrib(double A[F][C], double Aloc[F][2], MPI_Comm com)
{
  int p, rank, i;
  MPI_Datatype blq;
  MPI_Comm_size(com, &p);
  MPI_Comm_rank(com, &rank);
  MPI_Type_vector(F, 2, C, MPI_DOUBLE, &blq);
  MPI_Type_commit(&blq);
  if (rank==0) {
    for (i=1; i<p; i++) {
      MPI_Send(&A[0][i*2], 1, blq, i, 33, com);
    }
    MPI_Sendrecv(&A[0][0], 1, blq, 0, 33, &Aloc[0][0], F*2, MPI_DOUBLE,
                 0, 33, com, MPI_STATUS_IGNORE);
  }
  else {
    MPI_Recv(&Aloc[0][0], F*2, MPI_DOUBLE, 0, 33, com, MPI_STATUS_IGNORE);
  }
  MPI_Type_free(&blq);
}
```

Cuestión 3–15

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: $A = \begin{pmatrix} * & & & & * \\ & * & & * & \\ & & * & & \\ & * & & * & \\ * & & & & * \end{pmatrix}$

(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]) {
```

Solución:

```
void sendrecv_diagonals(double A[N][N]) {
  MPI_Datatype principal,antidiag;
  MPI_Type_vector(N, 1, N+1, MPI_DOUBLE, &principal);
  MPI_Type_commit(&diagonal);
  MPI_Type_vector(N, 1, N-1, MPI_DOUBLE, &antidiag);
  MPI_Type_commit(&antidiag);
  MPI_Bcast(A, 1, principal, 0, MPI_COMM_WORLD);
  MPI_Bcast(&A[0][N-1], 1, antidiag, 0, MPI_COMM_WORLD);
  MPI_Type_free(&principal);
  MPI_Type_free(&antidiag);
}
```

(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]) {
```

Solución:

```
void sendrecv_diagonals(double A[N][N], double prin[N], double anti[N]) {
  int nprocs, id, p;
  MPI_Datatype principal,antidiag;
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  if (id==0) {
    MPI_Type_vector(N, 1, N+1, MPI_DOUBLE, &principal);
    MPI_Type_commit(&diagonal);
    MPI_Type_vector(N, 1, N-1, MPI_DOUBLE, &antidiag);
    MPI_Type_commit(&antidiag);
    MPI_Sendrecv(A, 1, principal, 0, 0, prin, N ,MPI_DOUBLE, 0, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv(&A[0][N-1], 1, antidiag, 0, 1, anti, N,
                 MPI_DOUBLE, 0, 1, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    for (p=1; p<nprocs; p++) {
      MPI_Send(A, 1, principal, p, 0, MPI_COMM_WORLD);
      MPI_Send(&A[0][N-1], 1, antidiag, p, 1, MPI_COMM_WORLD);
    }
```

```
    MPI_Type_free(&principal);
    MPI_Type_free(&antidiag);
  }
  else {
    MPI_Recv(prin, N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(anti, N, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }
}
```

La solución anterior se puede simplificar mediante el uso de la primitiva `MPI_Bcast`:

```
void sendrecv_diagonals(double A[N][N], double prin[N], double anti[N]) {
  int id;
  MPI_Datatype principal,antidiag;
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  if (id==0) {
    MPI_Type_vector(N, 1, N+1, MPI_DOUBLE, &principal);
    MPI_Type_commit(&diagonal);
    MPI_Type_vector(N, 1, N-1, MPI_DOUBLE, &antidiag);
    MPI_Type_commit(&antidiag);
    MPI_Sendrecv(A, 1, principal, 0, 0, prin, N ,MPI_DOUBLE, 0, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv(&A[0][N-1], 1, antidiag, 0, 1, anti, N,
                 MPI_DOUBLE, 0, 1, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    MPI_Type_free(&principal);
    MPI_Type_free(&antidiag);
  }
  MPI_Bcast(prin, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast(anti, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

Cuestión 3–16

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])
```

Solución:

```
int MPI_Reparte_col_cic(float mat[M][N], float lmat[M][N/4])
{
   int me, size, i;
   MPI_Datatype col;

   MPI_Comm_rank(MPI_COMM_WORLD, &me);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   if (size!=4) {
      printf("Error, el número de procesos debe ser 4\n");
      return 1;
   }

   MPI_Type_vector(M*N/4,1,4,MPI_FLOAT,&col);
   MPI_Type_commit(&col);
   if (me==0) {
      for (i=1;i<size;i++)
         MPI_Send(&mat[0][i],1,col,i,0,MPI_COMM_WORLD);
      MPI_Sendrecv(mat,1,col,0,0,lmat,M*N/4,MPI_FLOAT,0,0,MPI_COMM_WORLD,
                   MPI_STATUS_IGNORE);
   } else
      MPI_Recv(lmat,M*N/4,MPI_FLOAT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

   MPI_Type_free(&col);
   return 0;
}
```