1. Comunicación punto a punto

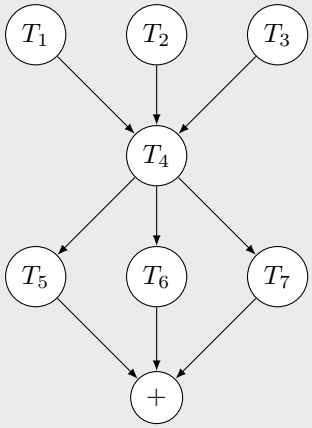
Cuestión 1–1

Dado el siguiente código secuencial:

```
double calculo(int i,int j,int k)
{
  double a,b,c,d,x,y,z;
  a = T1(i);
  b = T2(j);
  c = T3(k);
  d = T4(a+b+c);
  x = T5(a/d);
  y = T6(b/d);
  z = T7(c/d);
  return x+y+z;
}
```

(a) Dibuja el grafo de dependencias de las diferentes tareas, incluyendo el cálculo del resultado de la función.

Solución: El grafo de dependencias es el siguiente:



La última tarea representa la suma necesaria en la instrucción `return`. Existen tres dependencias ($T_1 \rightarrow T_5$, $T_2 \rightarrow T_6$ y $T_3 \rightarrow T_7$) que no se han representado, ya que se satisfacen indirectamente a través de las dependencias con T_4.

(b) Realiza una implementación paralela en MPI suponiendo que el número de procesos es igual a 3.

Solución: En la primera y tercera fase del algoritmo los tres procesos están activos, mientras que en la segunda fase sólo uno de ellos (por ejemplo P_0) ejecuta tarea. Antes de T_4, P_0 tiene que recibir `b` y `c` de los otros procesos, y al finalizar T_4 se debe enviar el resultado `d` a P_1 y P_2. Finalmente, hay que sumar las variables `x`, `y`, `z`, por ejemplo en el proceso P_0. **Nota**: en la solución propuesta,

el valor de retorno únicamente es válido en P_0. Si quisiéramos que todos los procesos tuvieran el valor de retorno correcto habría que hacer comunicaciones adicionales.

```
double calculo(int i,int j,int k)
{
  double a,b,c,d,x,y,z;
  int p,rank;
  MPI_Comm_size(MPI_COMM_WORLD,&p);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  /* primera fase */
  if (rank==0) {
    a = T1(i);
    MPI_Recv(&b, 1, MPI_DOUBLE, 1, 111, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(&c, 1, MPI_DOUBLE, 2, 111, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else if (rank==1) {
    b = T2(j);
    MPI_Send(&b, 1, MPI_DOUBLE, 0, 111, MPI_COMM_WORLD);
  } else { /* rank==2 */
    c = T3(k);
    MPI_Send(&c, 1, MPI_DOUBLE, 0, 111, MPI_COMM_WORLD);
  }
  /* segunda fase */
  if (rank==0) {
    d = T4(a+b+c);
  }
  /* tercera fase */
  if (rank==0) {
    MPI_Send(&d, 1, MPI_DOUBLE, 1, 112, MPI_COMM_WORLD);
    MPI_Send(&d, 1, MPI_DOUBLE, 2, 112, MPI_COMM_WORLD);
    x = T5(a/d);
  } else if (rank==1) {
    MPI_Recv(&d, 1, MPI_DOUBLE, 0, 112, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    y = T6(b/d);
  } else { /* rank==2 */
    MPI_Recv(&d, 1, MPI_DOUBLE, 0, 112, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    z = T7(c/d);
  }
  /* suma final */
  if (rank==0) {
    MPI_Recv(&y, 1, MPI_DOUBLE, 1, 113, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(&z, 1, MPI_DOUBLE, 2, 113, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else if (rank==1) {
    MPI_Send(&y, 1, MPI_DOUBLE, 0, 113, MPI_COMM_WORLD);
  } else { /* rank==2 */
    MPI_Send(&z, 1, MPI_DOUBLE, 0, 113, MPI_COMM_WORLD);
  }
  return x+y+z;
}
```

(c) Implementa una versión modificada de forma que se realice la ejecución replicada de la tarea T_4, utilizando operaciones de comunicación colectiva.

Solución: La ejecución replicada de T_4 permite evitar la comunicación de la tercera fase. Para ello, al final de la primera fase hay que hacer que `a`, `b`, `c` estén disponibles en todos los procesos, o mejor aún que se haga una reducción-a-todos de dichas variables.

```
double calculo(int i,int j,int k)
{
  double a,b,c,d,*e,f,x,y,z;
  int p,rank;
  MPI_Comm_size(MPI_COMM_WORLD,&p);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  /* primera fase */
  if (rank==0) {
    a = T1(i);
    e = &a;
  } else if (rank==1) {
    b = T2(j);
    e = &b;
  } else { /* rank==2 */
    c = T3(k);
    e = &c;
  }
  MPI_Allreduce(e, &f, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  /* segunda fase */
  d = T4(f);
  /* tercera fase */
  if (rank==0) {
    x = T5(a/d);
  } else if (rank==1) {
    y = T6(b/d);
  } else { /* rank==2 */
    z = T7(c/d);
  }
  /* suma final */
  if (rank==0) {
    MPI_Recv(&y, 1, MPI_DOUBLE, 1, 113, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(&z, 1, MPI_DOUBLE, 2, 113, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else if (rank==1) {
    MPI_Send(&y, 1, MPI_DOUBLE, 0, 113, MPI_COMM_WORLD);
  } else { /* rank==2 */
    MPI_Send(&z, 1, MPI_DOUBLE, 0, 113, MPI_COMM_WORLD);
  }
  return x+y+z;
}
```

Cuestión 1–2

Dada la siguiente función:

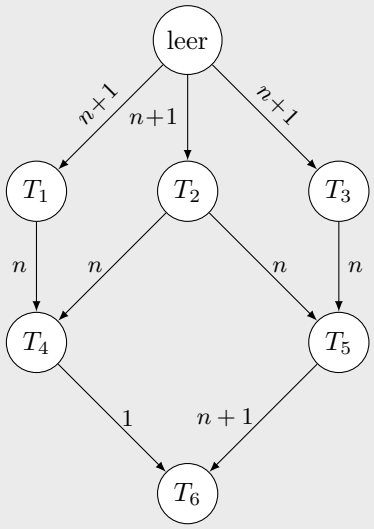
```
double funcion()
{
   int i,n,j;
   double *v,*w,*z,sv,sw,x,res=0.0;
```

```
    /* 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 T5 es igual al de T4, y el coste de T6 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.

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

   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, &status);

      /* 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, &status);
   }
   else if (rank==1) {
      w = (double*) malloc(n*sizeof(double));
      MPI_Recv(&n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
      MPI_Recv(w, n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);

      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, &status);
         }
         else if (rank==2) {
            z = (double*) malloc(n*sizeof(double));
            MPI_Recv(&n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
            MPI_Recv(z, n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);

            calcula_z(n,z);              /* tarea 3 */
            MPI_Recv(w, n, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &status);

            /* tarea 5 (mismo codigo del caso secuencial) */
            ...

            MPI_Recv(&sv, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);

            /* tarea 6 (mismo codigo del caso secuencial) */
            ...

            /* Enviar el resultado de la 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 1–3

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 procesadores 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 del vector v completo
 */
```

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 del vector v completo
 */
{
   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 mismo */
         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 1–4

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 1–5

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 1–6

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 1–7

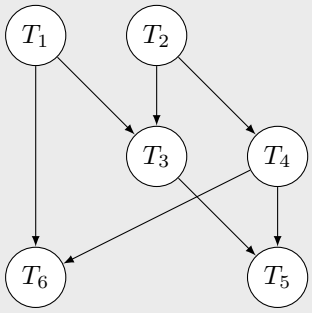
Se quiere paralelizar el siguiente código mediante MPI. Suponemos que se dispone de 3 procesos.

```
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 1–8

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 (si las dos respuestas son ligeras variaciones de la misma solución, se tendrá en cuenta solo una).

```
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, MPI_STATUS_IGNORE);
```

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, MPI_STATUS_IGNORE);
} else {
```

```
    MPI_Recv(rbuf, N, MPI_INT, src, 111, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    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, MPI_STATUS_IGNORE);
```

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, MPI_STATUS_IGNORE);
  MPI_Wait(&req, MPI_STATUS_IGNORE);
```

Cuestión 1–9

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];
```

Notas: 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 1–10

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 renvía 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 el esquema de comunicación previo 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, mi proceso izdo. es %d y mi proceso dcho. es %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 de un anillo de %d procesos: %f\n", size, (t2-t1)/REPS);
  }
  MPI_Finalize();
  return 0;
}
```

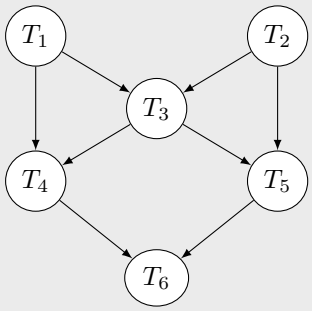
Cuestión 1–11

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 1–12

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.
- Se debe hacer sin 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_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, MPI_STATUS_IGNORE);
} else if (rank==proc2) {
  int i;
  MPI_Recv(x2, N, MPI_DOUBLE, proc1, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Send(x, N, MPI_DOUBLE, proc1, 100, MPI_COMM_WORLD);
  for (i=0; i<N; i++)
    x[i] = x2[i];
}
```

Cuestión 1–13

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 1–14

(a) Implementa 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

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) {
            for (j=0; j<N; j++)          /* Copia local de la fila */
               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.

2. Comunicación colectiva

Cuestión 2–1

El siguiente fragmento de código permite calcular el producto de una matriz cuadrada por un vector, ambos de la misma 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;
}
```

Cosas 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_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, MPI_STATUS_IGNORE);
  }
  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, MPI_STATUS_IGNORE);
  }
  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,MPI_STATUS_IGNORE);
        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. Faltaría cambiar `a` por `b` en las llamadas a `malloc`, `free` y `procesa`.

Cuestión 2–4

Desarrolla un programa usando 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("ERROR: 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:

```
si id=0
  n = inicial
  para i=0 hasta 100
    n = siguiente(n)
  fpara
  envia mensaje de aviso a procesos 1..np-1
  recibe m[k] de procesos 1..np-1
  n = combina(n,m[k]) para cada 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:

```
MPI_Comm_rank(MPI_COMM_WORLD, &id);
MPI_Comm_size(MPI_COMM_WORLD, &np);
n = inicial(id);
if (id==0) {
  for (i=0;i<100;i++) n = siguiente(n);
  for (k=1;k<np;k++) {
    MPI_Send(0, 0, MPI_INT, k, TAG_AVISO, MPI_COMM_WORLD);
  }
} else {
  MPI_Irecv(0, 0, MPI_INT, 0, TAG_AVISO, 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 el valor del número π:

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

hits = 0;
for (i=0; i<points; i++) {
  rx = (double)(rand_r(&seed))/((double)RAND_MAX);
  ry = (double)(rand_r(&seed))/((double)RAND_MAX);
  if (((rx-0.5)*(rx-0.5)+(ry-0.5)*(ry-0.5))<0.25) hits++;
}
computed_pi = 4.0*(double)hits/((double)points);
printf("Computed PI = %16.10lf\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`. Dado que todos los procesos reciben el valor de los argumentos de entrada, cada proceso solo tiene que calcular la cantidad de números aleatorios que debe generar.

```
double rx, ry, computed_pi;
long int i, points_per_proc, points, hits_per_proc, 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;
hits_per_proc = 0;
for (i=0; i<points_per_proc; i++) {
  rx = (double)(rand_r(&seed))/((double)RAND_MAX);
  ry = (double)(rand_r(&seed))/((double)RAND_MAX);
  if (((rx-0.5)*(rx-0.5)+(ry-0.5)*(ry-0.5))<0.25) hits_per_proc++;
}

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

if (!myproc) {
  computed_pi = 4.0*(double)hits/((double)points);
  printf("Computed PI = %16.10lf\n", computed_pi);
}
```

Cuestión 2–7

La infinito-norma de una matriz se define como el máximo de las sumas de los valores absolutos de los elementos de cada fila: $\max_{i=0}^{n-1} \left\{ \sum_{j=0}^{m-1} |a_{i,j}| \right\}$. 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,norm;

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

(a) Implementar una versión paralela mediante MPI utilizando operaciones colectivas en la medida de lo posible. 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,norm,normlcl;

  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);
  normlcl=fabs(Alocal[0][0]);
  for (i=0; i<N/p; i++) {
    s=0.0;
    for (j=0; j<N; j++)
      s+=fabs(Alocal[i][j]);
    if (s>normlcl)
      normlcl=s;
  }
  MPI_Reduce(&normlcl, &norm, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
  return norm;
}
```

(b) Obtener el coste computacional y de comunicaciones del algoritmo paralelo. 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) Calcular 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:

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

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

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

Solución:

```
MPI_Gather(z, LNZ, MPI_FLOAT, zfull, LNZ, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

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

```
long int factor, producto;
MPI_Allreduce(&factor, &producto, 1, MPI_LONG, MPI_PROD, MPI_COMM_WORLD);
```

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

Solución: Una posible implementación muy sencilla sería aquella en que el proceso 0 se encarga de realizar todos los productos.

```
#define TAG 999
long int factor, producto, tmp;
int i, k, sz, rank;
MPI_Comm_size(MPI_COMM_WORLD, &sz);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (!rank) {  /* el proceso 0 actua de root */
  producto = factor;
  for (k=1;k<sz;k++) {
    MPI_Recv(&tmp, 1, MPI_LONG, k, TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    producto *= tmp;
  }
  for (k=1;k<sz;k++) {
    MPI_Send(&producto, 1, MPI_LONG, k, TAG, MPI_COMM_WORLD);
  }
} else {
  MPI_Send(&factor, 1, MPI_LONG, 0, TAG, MPI_COMM_WORLD);
  MPI_Recv(&producto, 1, MPI_LONG, 0, TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Cuestión 2–11

(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 mediante `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 la matriz `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 2–12

La siguiente 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
$t(n,p) = 2(p-1)(t_s + \frac{n}{p}t_w) + \frac{2n}{p} + (p-1)(t_s + t_w) + p - 1 \approx 3p \cdot t_s + (2n+p)t_w + \frac{2n}{p}$
$S(n,p) = \frac{t(n)}{t(n,p)} = \frac{2n}{3p \cdot t_s + (2n+p)t_w + \frac{2n}{p}}$
$\lim_{n\to\inf} S(n,p) = \frac{2}{2 \cdot t_w + \frac{2}{p}} = \frac{2p}{2p \cdot t_w + 2}$
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 2–13

Dado el siguiente programa secuencial:

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

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

  printf("Número de iteraciones: ");
  scanf("%d",&n);

  min = 100000;
  for (i = 1; i <= n; i++) {
    val = calcula_valor(i);
    if (val < min) min = val;
  }
  printf("Mínimo: %d\n",min);

  return 0;
}
```

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

Solución:

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

  MPI_Init(&argc,&argv);

  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  MPI_Comm_size(MPI_COMM_WORLD,&np);

  if (myid == 0) {
    printf("Número de iteraciones: ");
    scanf("%d",&n);
    for (i = 1; i < np; i++)
      MPI_Send(&n,1,MPI_INT,i,2010,MPI_COMM_WORLD);
  } else
    MPI_Recv(&n,1,MPI_INT,0,2010,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

  min = 100000;
  for (i = myid+1; i <= n; i+=np) {
    val = calcula_valor(i);
    if (val < min)  min = val;
  }

  if (myid == 0) {
    for (i = 1; i < np; i++) {
      MPI_Recv(&val,1,MPI_INT,i,1492,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
      if (val < min)  min = val;
    }
```

```
        printf("Mínimo: %d\n",min);
      } else
        MPI_Send(&min,1,MPI_INT,0,1492,MPI_COMM_WORLD);

      MPI_Finalize();
      return 0;
    }
```

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

Solución: Se cambiarían los dos `if (myid==0)` respectivamente por

```
      if (myid == 0) {
        printf("Número de iteraciones: ");
        scanf("%d",&n);
      }
      MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
```

y por

```
      MPI_Reduce(&min,&val,1,MPI_INT,MPI_MIN,0,MPI_COMM_WORLD);
      if (myid == 0)
        printf("Mínimo: %d\n",val);
```

Cuestión 2–14

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

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

3. Tipos de datos

Cuestión 3–1

Supongamos definida una matriz de enteros `A[N][N]`. Define un tipo derivado MPI y realiza las correspondientes llamadas para el envío desde P_0 y la recepción en P_1 de un dato de ese tipo, en los siguientes casos:

(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. Con tipo derivado `vector` de MPI sería así:

```
int A[N][N];
MPI_Status status;
MPI_Datatype newtype;

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

Una solución equivalente se obtendría con `MPI_Type_contiguous`. En este caso, no es realmente necesario crear un tipo MPI, por estar los elementos contiguos en memoria:

```
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, &status);
}
```

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

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

```
int A[N][N];
MPI_Status status;
MPI_Datatype newtype;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Type_vector(N, 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, &status);
}
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_datos(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 en lugar de tres.

Solución:

```
void distribuye_datos(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);

  MPI_Datatype Tnuevo;
  int longitudes[]={1,n,n};
  MPI_Datatype tipos[]={MPI_INT, MPI_INT, MPI_DOUBLE};
  MPI_Aint despls[3];
  MPI_Aint dir1, dirx, diry, dira;

  /* 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_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);
  }
}
```

(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 $\star$:

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

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

Solución: Creamos un vector de 3 elementos, cada uno de ellos con longitud 3 y con separación `N`.

```
double A[N][N];
MPI_Datatype newtype;
```

```
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);
```

Cuestión 3–5

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

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

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

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

Solución:

```
void comunica_vector(double xloc[], int n, int p, int rank, double x[])
/* rank es el indice del proceso local                                  */
/* Esta funcion asume que n es múltiplo exacto de p                     */
{
   int rank2, nloc=n/p, j;
   MPI_Datatype Tnuevo;

   MPI_Type_vector(nloc, 1, p, MPI_DOUBLE, &Tnuevo);
   MPI_Type_commit(&Tnuevo);

   if (rank==0) {
      for (rank2=1; rank2<p; rank2++) {
         MPI_Recv(&x[rank2], 1, Tnuevo, rank2, 123, MPI_COMM_WORLD,
               MPI_STATUS_IGNORE);
      }
      for (j=0; j<nloc; j++)
         x[j*p] = xloc[j];
   } else {
      MPI_Send(xloc, nloc, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD);
   }
   MPI_Type_free(&Tnuevo);
}
```

Otra solución, aunque menos eficiente, sería utilizar un array auxiliar en vez de tipos de datos derivados:

```
void comunica_vector(double xloc[], int n, int p, int rank, double x[])
/* rank es el indice del proceso local                                  */
```

```
/* Esta funcion asume que n es múltiplo exacto de p                    */
{
   int rank2, nloc=n/p, j;
   double *xaux = (double *) malloc(nloc*sizeof(double));

   if (rank==0) {
      for (rank2=1; rank2<p; rank2++) {
         MPI_Recv(xaux, nloc, MPI_DOUBLE, rank2, 123, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
         for (j=0; j<nloc; j++)
            x[j*p+rank2] = xaux[j];
      }
      for (j=0; j<nloc; j++)
         x[j*p] = xloc[j];
   } else {
      MPI_Send(xloc, nloc, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD);
   }
   free(xaux);
}
```

Cuestión 3–6

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 */

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 3–7

Dado el siguiente fragmento de código de un programa paralelo:

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

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

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

Solución:

```
MPI_Datatype dtype;
MPI_Type_vector(NFIL, 1, NCOL, MPI_DOUBLE, &dtype);
MPI_Type_commit(&dtype);
if (rank==0) {
   MPI_Send(&A[0][j], 1, dtype, proc, 100, MPI_COMM_WORLD);
   MPI_Recv(&A[0][j], 1, dtype, proc, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
} else if (rank==proc) {
   int i;
   MPI_Recv(col, NFIL, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   for (i=0; i<NFIL; i++)
      col[i] = -col[i];
   MPI_Send(col, NFIL, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD);
}
```

Cuestión 3–8

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 así:

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

Deberás 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 3–9

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 3–10

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);
}
```