# Índice



# 1. Memoria Compartida y OpenMP

## 1.1. Paralelismo de bucles

**Cuestión 1.1–1**

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

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

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

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

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

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

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

**Solución:**

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

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

  #pragma omp parallel for private(j,s)
  for (i=0; i<n; i++) {
    s = 0;
    for (j=0; j<n; j++)
      s += fabs(A[i][j]);
    if (s>norm)
      #pragma omp critical
      if (s>norm)
        norm = s;
  }
  return norm;
}
```

La paralelización la situamos en el bucle más externo para una mayor granularidad y menor tiempo de sincronización. Teniendo en cuenta que todas las iteraciones tienen el mismo coste computacional, se puede realizar una distribución estática de las iteraciones (planificación por defecto).

La paralelización genera una condición de carrera en la actualización del máximo de las sumas por filas. Para evitar errores en las actualizaciones simultáneas se utiliza una sección crítica que evita la modificación concurrente de `norm`. Para evitar una excesiva secuencialización se incluye la sección crítica dentro de la comprobación del máximo, repitiéndose la comprobación dentro de la sección crítica para asegurar que sólo se modifica el valor de `norm` si es un valor superior al acumulado.

(b) Calcula el coste computacional (en flops) de la versión original secuencial y de la versión paralela desarrollada.
**Nota**: Se puede asumir que la dimensión de la matriz $n$ es un múltiplo exacto del número de hilos $p$.
**Nota 2**: Se puede asumir que el coste de la función `fabs` es de 1 Flop.

**Solución:** $t(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 2 = 2n^2$ flops $\qquad t(n,p) = \sum_{i=0}^{\frac{n}{p}-1}\sum_{j=0}^{n-1} 2 = 2\frac{n^2}{p}$ flops

No se incluye el coste derivado de las operaciones de sincronización porque dependen de los valores de la matriz y las condiciones de ejecución.

(c) Calcula la máxima aceleración (speedup) posible que puede obtenerse con el código paralelo. Calcula también el valor de la eficiencia.

**Solución:** $S(n,p) = \frac{t(n)}{t(n,p)} = p \qquad E(n,p) = \frac{S(n,p)}{p} = 1$

La aceleración es la máxima esperable y la eficiencia es del 100 %, lo que indica un aprovechamiento

máximo de los procesadores.

## 1.2. Regiones paralelas

**Cuestión 1.2–1**

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

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

**Solución:**

```
int busqueda(int x[], int n, int valor)
{
   int encontrado=0, i, salto;
   #pragma omp parallel private(i)
   {
      i = omp_get_thread_num();
      salto = omp_get_num_threads();
      while (!encontrado && i<n) {
         if (x[i]==valor) encontrado=1;
         i += salto;
      }
   }
   return encontrado;
}
```

**Cuestión 1.2–2**

Dado un vector $v$ de $n$ elementos, la siguiente función calcula su 2-norma $\|v\|$, definida como:

$$\|v\| = \sqrt{\sum_{i=1}^{n} v_i^2}$$

```
double norma(double v[], int n)
{
   int i;
   double r=0;
   for (i=0; i<n; i++)
      r += v[i]*v[i];
   return sqrt(r);
}
```

(a) Paralelizar la función anterior mediante OpenMP, siguiendo el siguiente esquema:

- En una primera fase, se quiere que cada hilo calcule la suma de cuadrados de un bloque de $n/p$ elementos del vector $v$ (siendo $p$ el número de hilos). Cada hilo dejará el resultado en la posición correspondiente de un vector `sumas` de $p$ elementos. Se puede asumir que el vector `sumas` ya ha sido creado (aunque no inicializado).
- En una segunda fase, uno de los hilos calculará la norma del vector, a partir de las sumas parciales almacenadas en el vector `sumas`.

**Solución:**

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

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

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

(b) Paralelizar la función de partida mediante OpenMP, usando otra aproximación distinta de la del apartado anterior.

**Solución:**

```
double norma(double v[], int n)
{
   int i;
   double r=0;
   #pragma omp parallel for reduction(+:r)
   for (i=0; i<n; i++)
      r += v[i]*v[i];
   return sqrt(r);
}
```

(c) Calcular el coste a priori del algoritmo secuencial de partida. Razonar cuál sería el coste del algoritmo paralelo del apartado a, y el speedup obtenido.

**Solución:** Coste del algoritmo secuencial (el coste de la raíz cuadrada es despreciable frente al coste del bucle `i`):

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

Coste del algoritmo paralelo: como cada iteración del bucle `i` cuesta 2 flops y cada hilo realiza $n/p$ iteraciones, el coste es de $t(n,p) = 2n/p$ flops. El speedup es de $S(n,p) = 2n/(2n/p) = p$.

**Cuestión 1.2–3**

Dada la siguiente función:

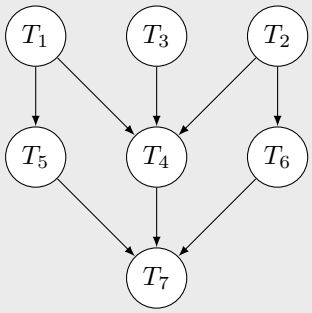
```
double funcion(int n, double u[], double v[], double w[], double z[])
{
  int i;
  double sv,sw,res;

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

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

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

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

**Solución:** El grafo de dependencias se muestra a continuación:




(b) Paraleliza la función de forma eficiente.

**Solución:** La paralelización se puede realizar con secciones y las dependencias se pueden implementar con barreras implícitas. Todas las variables son compartidas excepto el contador de los bucles. El código paralelo es:

```
double funcion(int n, double u[], double v[], double w[], double z[])
{
```

```
    int i;
    double sv,sw,res;

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

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

**Solución:** Suponiendo que el coste de cada tarea `calcula_X` es 1, el coste total del algoritmo secuencial sería 4. A la vista del grafo de dependencias se deduce que el grado de concurrencia es 3 (el máximo número de tareas que se pueden ejecutar en paralelo es 3). Las tres primeras tareas `calcula_X` se pueden ejecutar en paralelo, pero la última no. Por tanto, el coste del algoritmo paralelo será 2 en el caso de utilizar 3 o más procesadores. El speedup máximo en ese caso será $4/2=2$.

## 1.3. Sincronización

## 1.4. Tareas

**Cuestión 1.4–1**

Una sucesión de *Fibonacci* se define como la siguiente sucesión infinita:

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad 55 \quad \ldots$$

calculándose cada elemento de la serie así $f_n = f_{n-1} + f_{n-2}$, $\forall n > 1$, $f_0 = 0$ y $f_1 = 1$.

Una forma burda pero válida para obtener el elemento $f_n$ es utilizar la siguiente función:

```
int fibonacci(int n) {
   int i,j;
   if (n<2) {
     return n;
   } else {
     i = fibonacci(n-1);
     j = fibonacci(n-2);
     return i+j;
   }
}
```

Implementar una versión en OpenMP que permita su cálculo en paralelo en memoria compartida mediante tareas OpenMP (`task`).

**Solución:** La paralelización en OpenMP de la rutina anterior se realizaría como sigue:

```
int fibonacci(int n) {
   int i,j;
   if (n<2) {
     return n;
   } else {
     #pragma omp task shared(i) if(n<10)
     i = fibonacci(n-1);
     #pragma omp task shared(j) if(n<10)
     j = fibonacci(n-2);
     #pragma omp taskwait
     return i+j;
   }
}
```

y la llamada a la rutina en el programa principal se haría así:

```
#pragma omp parallel
#pragma omp single
m = fibonacci(n);
```

Dado que el coste de la generación de tareas es considerable y esto puede ralentizar el proceso en lugar de optimizarlo, se corta la generación de tareas si el valor de `n` es menor de cierta cantidad. Este es un caso en el que es necesario sincronizar las dos tareas generadas para poder utilizar el resultado devuelto y regresar así hacia arriba en el árbol de llamadas con el dato correcto.

# 2. Paso de Mensajes y MPI

## 2.1. Comunicación punto a punto

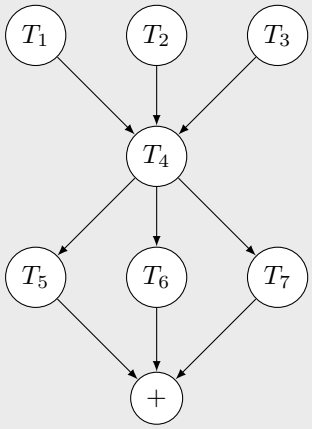
**Cuestión 2.1–1**

Dado el siguiente código secuencial:

```
int calculo(int i,int j,int k)
{
  double a,b,c,d;
  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 ($T_1 \to T_5$, $T_2 \to T_6$ y $T_3 \to T_7$) 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 $P_0$.

```
int calculo(int i,int j,int k)
{
  double a,b,c,d;
  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.

```
int calculo(int i,int j,int k)
{
  double a,b,c,d,*e,f;
  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 2.1–2**

Dada la siguiente función:

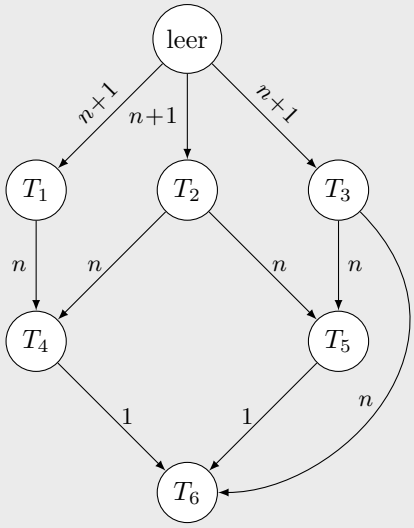
```
double funcion()
{
   int i,n,j;
   double *v,*w,*z,sv,sw,x,res;

   /* 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++) w[i]=sw*w[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 de cada una 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 3 las tareas 3, 4 y 5.

```
double funcion()
{
   int i,n,j;
   double *v,*w,*z,sv,sw,x,res;
   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) {
      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) {
      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 2.1–3**

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 2.1–4**

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.

(a) 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);
  }
}
```

(b) Suponiendo que cada tarea tiene un coste de $2n^2$ flops, calcular el coste secuencial y paralelo del algoritmo para $m$ vectores y $T$ tareas suponiendo que todos los nodos están interconectados entre sí y que el coste del mensaje testigo de terminación es despreciable.

**Solución:** El coste secuencial del algoritmo es de $2n^2mT$ flops.

Para calcular el coste paralelo obtenemos los costes aritmético ($t_a$) y de comunicaciones ($t_c$) por separado.

La manera más intuitiva de calcular el coste paralelo de un pipeline consiste en analizar:

- El coste de cálculo del primer elemento a procesar, en este caso, el primer vector. Este coste es el coste por tarea ($2n^2$) multiplicado por el número de tareas $T$, $2n^2T$.
- Una vez haya salido el primer elemento del cauce, se obtiene un nuevo elemento cada $2n^2$ flops que, multiplicado por el número de elementos ($m$) menos 1 daría el coste aritmético asociado a los $m-1$ elementos restantes.

El coste aritmético, por tanto, es

$$t_a = 2n^2T + 2n^2(m-1) = 2n^2(T+m-1)\ .$$

Para obtener el coste de comunicaciones se sigue un razonamiento similar. El primer mensaje tiene que dar $p$ "saltos" desde que sale del proceso maestro hasta que vuelve a él. Por lo tanto, este coste es de

$$p \cdot (t_s + n \cdot t_w)\ .$$

A partir de entonces, el proceso maestro recibe $m-1$ mensajes de tamaño `n`. Se ha supuesto que los nodos están todos interconectados entre sí, por lo que se asume que pueden darse comunicaciones concurrentes. El coste de comunicaciones total en función del número de tareas ($T$) es

$$t_c = (T+1)\cdot(t_s + n\cdot t_w) + (m-1)\cdot(t_s + n\cdot t_w) = (T+m)\cdot(t_s + n\cdot t_w)\ .$$

El coste total del algoritmo paralelo sería:

$$t_a + t_c = (T+m-1)2n^2 + (T+m)(t_s + nt_w)\ .$$

(c) Obtener el Speedup del algoritmo paralelo. Analizar dicho Speedup cuando el número de datos ($m$) se hace grande. En tal caso, analizad también cuál sería el Speedup cuando el tamaño del vector ($n$) se hace grande. Por último, para $m$ y $n$ suficientemente grandes indicad la eficiencia.

**Solución:** El Speedup se calcula así

$$S = \frac{2n^2mT}{(T+m-1)2n^2 + (T+m)(t_s + nt_w)}\ .$$

Para analizar el Speedup cuando $m$ se hace grande vemos qué pasa al tender $m$ a infinito:

$$\lim_{m\to\infty} S = \lim_{m\to\infty} \frac{2n^2T}{(\frac{T-1}{m}+1)2n^2 + (\frac{T}{m}+1)(t_s+nt_w)} = \frac{2n^2T}{2n^2 + (t_s+nt_w)} = \frac{T}{1+\frac{t_s+nt_w}{2n^2}}\ .$$

La expresión anterior dice que, para un número suficientemente grande de vectores a procesar, el Speedup es proporcional al número de tareas del pipeline, es decir, tendremos más incremento de velocidad cuanto más tareas haya en el pipeline.
Sin embargo, el Speedup sigue limitado por el coste de las comunicaciones. Si utilizamos vectores suficientemente grandes, es decir, cuando $n$ tiende a infinito, obtendremos lo siguiente:

$$\lim_{m,n\to\infty} S = \frac{T}{1+\frac{t_s+nt_w}{2n^2}} = T\ ,$$

lo que viene a indicar que se tendrá más incremento de velocidad cuanto mayor sea tanto el número de vectores a procesar como el tamaño de los mismos y que éste será proporcional al número de tareas.
Para obtener la eficiencia en el caso de que tanto $m$ como $n$ sean grandes (infinito) utilizaremos la última expresión del Speedup:

$$E = \frac{T}{p} = \frac{p-1}{p}\ .$$

Esta eficiencia se aproxima al 100 % con el número de procesos. Para obtener una eficiencia máxima del 100 % podríamos, por ejemplo, asignar una tarea a $P_0$ o bien simplemente mapear $P_0$ y $P_1$ en el mismo procesador, ya que $P_0$ no hace trabajo efectivo.

**Cuestión 2.1–5**

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 2.1–6**

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) Realiza una implementación siguiendo el modelo maestro-esclavo de dicho algoritmo utilizando MPI.

**Solución:**

```
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);
    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);

      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);
    }
```

(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 $2 \cdot n \cdot t_s + t_w \cdot (9 \cdot n + m)$.

## 2.2. Comunicación colectiva

**Cuestión 2.2–1**

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 2.2–2**

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 2.2–3**

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 = malloc(nb*sizeof(int));

  /* Repartir la a entre todos los procesos (2) */
  if (!p) {
    for (i=1; i<np; i++)
      MPI_Send(&a[i*nb], nb, MPI_INT, i, 25, MPI_COMM_WORLD);
  } else {
```

```
      MPI_Recv(a, nb, MPI_INT, 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 2.2–4**

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 = 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 2.2–5**

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);
for (k=1;k<np;k++) n = combina(n,m[k]);
```

**Cuestión 2.2–6**

Dado el siguiente fragmento de programa que calcula el valor del número $\pi$:

```
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);
}
```

## 2.3. Tipos de datos

**Cuestión 2.3–1**

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 2.3–2**

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 */
  dir1=MPI_Get_address(datos, &dir1);
  dirx=MPI_Get_address(&(datos->x), &dirx);
  diry=MPI_Get_address(&(datos->y[0]), &diry);
  dira=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 2.3–3**

Supóngase que se desea implementar la versión estática del problema del Sudoku. El proceso 0 genera una matriz $A$ de tamaño $n \times 81$, siendo $n$ el número de tableros que hay en la matriz.

(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_TAREA` que represente la estructura anterior.

**Solución:**

```
MPI_Datatype MPI_TAREA;
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, &MPI_TAREA);
MPI_Type_commit(&MPI_TAREA);
```

**Cuestión 2.3–4**

El siguiente fragmento de código implementa el cálculo del cuadrado de una matriz triangular superior con almacenamiento compacto (se evita almacenar los ceros). La matriz del ejemplo sería:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 5 & 6 & 7 \\ 0 & 0 & 8 & 9 \\ 0 & 0 & 0 & 10 \end{bmatrix}$$

```
#define DIMN 4
#define triang(i,j,n) ( (i)*(n) + ((i)*((i)-1))/2 + (j) - (i) )

double A[(DIMN*DIMN+DIMN)/2] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
double A2[(DIMN*DIMN+DIMN)/2] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
int i, j, k, n = DIMN;

for (i=0; i<n; i++)
  for (j=i; j<n; j++)
    for (k=i; k<=j; k++)
       A2[triang(i,j,n)] += A[triang(i,k,n)]*A[triang(k,j,n)];
```

(a) Escribe el código MPI para distribuir la matriz triangular superior (asumir que la matriz es cuadrada), mediante comunicación punto a punto y `MPI_Pack`. Justifica si sería más apropiado utilizar una distribución de bloques de filas o una distribución cíclica por filas (se puede justificar calculando el coste para cada una de las distribuciones en el caso de ejemplo propuesto 4x4 con por ejemplo 2 procesos).

**Solución:** La distribución cíclica por filas es mejor que la distribución por bloques de filas. Esto se puede ver tomando el ejemplo dado:

1. Distribución por bloques de filas. $P_0$ obtiene dos filas con 4 y 3 elementos (total 7) y $P_1$ obtiene dos filas con 2 y 1 elementos (total 3). La media sería 5, y la desviación estándar es 2.

2. Distribución cíclica por filas. $P_0$ obtiene dos filas con 4 y 2 elementos (total 6) y $P_1$ obtiene dos filas con 3 y 1 elementos (total 4). La media sería 5, y la desviación estándar es 1.

```
if (rank==0) {
  for (i=1;i<p;i++) {
    pos = 0;
    for (j=0;j<n/p;j++) {
      MPI_Pack(&A[triang(j*p+i,j*p+i,n)], n-(j*p+i),
               MPI_DOUBLE, buffer, bufsize, &pos, MPI_COMM_WORLD);
    }
    MPI_Send(buffer, pos, MPI_PACKED, i, 0, MPI_COMM_WORLD);
  }
} else {
  MPI_Recv(buffer, bufsize, MPI_PACKED, 0, 0, MPI_COMM_WORLD, &status);
  pos = 0;
  for (j=0;j<n/p;j++) {
    MPI_Unpack(buffer, bufsize, &pos, &A[triang(j*p+rank,j*p+rank,n)],
               n-(j*p+rank), MPI_DOUBLE, MPI_COMM_WORLD);
  }
}
```

(b) ¿Cuál sería el tiempo de comunicaciones de dicho programa?

**Solución:** En total hay $p-1$ mensajes, con tiempo de establecimiento $(p-1)t_s$, que enviarán la matriz completa ($(n^2+n)/2$ elementos), excepto la parte asociada a $P_0$ ($n+n-p+n-2\cdot p+... = \sum_{i=0}^{\frac{n}{p}-1}(n-i\cdot p) = \frac{n^2}{2\cdot p}$). La distribución cíclica garantiza que el número de elementos recibidos por cada proceso será más o menos el mismo, aproximadamente igual a $n^2/(2\cdot p)$. Por tanto, el coste de comunicación será:

$$t_c(n,p) = (p-1)\cdot\left(t_s + \frac{n^2}{2\cdot p}t_w\right)$$

(c) Implementar utilizando MPI la recogida de los resultados para que la matriz final esté en $P_0$.

**Solución:**

```
if (rank==0) {
  for (i=1;i<p;i++) {
    MPI_Recv(buffer, bufsize, MPI_PACKED, i, 0, MPI_COMM_WORLD, &status);
    pos = 0;
    for (j=0;j<n/p;j++) {
      MPI_Unpack(buffer, bufsize, &pos, &A2[triang(j*p+i,j*p+i,n)],
                 n-(j*p+i), MPI_DOUBLE, MPI_COMM_WORLD);
    }
  }
} else {
  pos = 0;
  for (j=0;j<n/p;j++) {
    MPI_Pack(&A2[triang(j*p+rank,j*p+rank,n)], n-(j*p+rank),
             MPI_DOUBLE, buffer, bufsize, &pos, MPI_COMM_WORLD);
  }
  MPI_Send(buffer, pos, MPI_PACKED, 0, 0, MPI_COMM_WORLD);
}
```

## 2.4. Comunicadores y topologías

**Cuestión 2.4–1**

En un programa MPI, se pretende definir dos comunicadores de manera que en cada comunicador se realice un procesamiento distinto. Si `p` es el número de procesos, define los comunicadores de manera que cada uno de ellos contenga la mitad de los procesos (si fuese impar `p`, el primer comunicador tendría uno más).

**Solución:**

```
int p, p1, p2, i;
int *ranks1, *ranks2;

MPI_Status status;
MPI_Comm new_comm1, new_comm2;
MPI_Group group_world, new_group1, new_group2;

MPI_Comm_size(MPI_COMM_WORLD,&p);
p1 = p/2;
p2 = p-p1;
ranks1 = (int*)malloc(p1*sizeof(int));
ranks2 = (int*)malloc(p2*sizeof(int));

for (i=0; i<p1; i++) ranks1[i] = i;
for (i=p1; i<p; i++) ranks2[i-p1] = i;

MPI_Comm_group(MPI_COMM_WORLD, &group_world);
MPI_Group_incl(group_world, p1, ranks1, &new_group1);
MPI_Comm_create(MPI_COMM_WORLD, new_group1, &new_comm1);
MPI_Group_incl(group_world, p2, ranks2, &new_group2);
MPI_Comm_create(MPI_COMM_WORLD, new_group2, &new_comm2);
```