Laboratorio de Paralelismo

MPI: ejemplos y ejercicios

> Máquinas y directorios

Vamos a trabajar con un *cluster* sencillo de 35 nodos: 32 nodos uniprocesador (Pentium IV, 3 GHz, 1,5 GB RAM, 2 MB cache) y tres nodos SMP de 4 procesadores cada uno, conectados mediante Gigabit Ethernet y un *switch* en una red local. Aunque no es un sistema de grandes prestaciones, nos permite ejecutar todo tipo de aplicaciones paralelas mediante paso de mensajes, analizar su comportamiento, etc.
Uno de los nodos del *cluster* hace las veces de servidor de ficheros y de punto de entrada al *cluster*; su dirección externa es **`g000002.gi.ehu.es`**, y su dirección en el *cluster* es `nodo00`. El resto de los nodos, `nodo01`, `nodo02`... `nodo31`, `acpt48`, `acpt49`, `acpt51` sólo son accesibles desde el `nodo00` (no tienen conexión al exterior, salvo los nodos SMP).

Cada grupo tiene abierta una cuenta en el *cluster*, de nombre **`clapxx`**, donde `xx` es el mismo número de cuenta que en el caso de las cuentas `mlap` en la máquina SMP. Por ejemplo (linux):

```
> ssh –l clap01 g000002.gi.ehu.es
```

Igual que en el caso anterior, en cada cuenta hay un directorio (`LPAR`) en el que iremos dejando los ejemplos, ejercicios... que vamos a utilizar. NO TRABAJÉIS EN ESA CARPETA, porque la iremos modificando a lo largo del curso.

Para poder ejecutar aplicaciones utilizando diferentes máquinas necesitamos que el sistema pueda entrar en las mismas usando `ssh` pero sin que se pida *password*. Para ello, hay que haber entrado una primera vez en cada máquina, para que nos reconozca como "usuarios autorizados". Hay que ejecutar los siguientes comandos:

- En el directorio de entrada, ejecutad:

```
> ssh-keygen -t rsa
```
(responded con `return` las tres veces)

con lo que se genera el directorio `.ssh` con los ficheros con claves `id.rsa` e `id_rsa.pub`

- Pasad al directorio `.ssh` y ejecutad:

```
> cp id_rsa.pub authorized_keys
> chmod go-rw authorized_keys
```

- A continuación, entrad y salid, una por una, en las máquinas del *cluster*:

```
> ssh nodoxx        (xx = 01 ... 31)
     (yes)
> exit
```

> MPICH2

Vamos a utilizar la versión **MPICH2** de MPI (es de libre distribución; si queréis, la podéis instalar en vuestro PC; otra implementación de gran difusión es LAM / Open MPI).
Previo a ejecutar programas en paralelo, hay que:

- Crear un fichero de nombre `.mpd.conf` que contenga una línea con el siguiente contenido:

  `secretword=palabra_secreta` (cualquier palabra)

  y cambiarle las protecciones:

  ```
  > chmod 600 .mpd.conf
  ```

- Poner en marcha el "entorno" MPICH2 (unos *daemons* mpd que se van a ejecutar en cada máquina del *cluster*):

  ```
  > mpdboot -v -n num_proc -f fichero_maquinas
  ```

  `fichero_maquinas` contiene los "nombres" de las máquinas del *cluster* con las que se quiere trabajar (por defecto se utiliza el fichero `mpd.hosts`).

  Otros comandos útiles:

`> mpdtrace`	devuelve las máquinas "activas"
`> mpdlistjobs`	lista los trabajos en ejecución en el anillo de *daemons*
`> mpdringtest 2`	recorre el anillo de *daemons* (2 veces) y devuelve el tiempo
`> mpd &`	lanza un solo *daemon* en el procesador local
`> mpd –help`	información del comando

  Si ha habido algún problema con la generación de *daemons*, etc., ejecutar `mpdcleanup` y repetir el proceso.

- A continuación podemos compilar y ejecutar programas:

  ```
  > mpicc [-Ox] -o pr1 pr1.c
  ```
  [x=1-3 nivel de optimiz.]

  ```
  > mpiexec -n xx pr1
  ```
  `xx` = número de procesos

  ejecuta el programa `pr1` en `xx` procesadores.

  ```
  > mpiexec -n 1 -host nodo00 p1 : -n 1 -host nodo01 p2
  > mpiexec -configfile procesos
  ```

  lanza dos programas, `p1` y `p2`, en los nodos 00 y 01, o bien tal como se especifica en el fichero `procesos`.

- Finalmente, para terminar una sesión de trabajo ejecutamos:

  ```
  > mpdallexit
  ```

(para más información sobre estos comandos: `comando --help`)

> JUMPSHOT

`Jumpshot` es la herramienta de análisis de la ejecución de programas paralelos que se distribuye junto con MPICH. Es una herramienta compleja, que permite analizar gráficamente ficheros de trazas (tipo "`log`") obtenidos de la ejecución de programas MPI a los que previamente se les ha añadido una serie de llamadas a MPE para recoger información.

Para generar el fichero `log` podemos añadir puntos de muestreo en lugares concretos (añadiendo funciones de MPE), o, en casos sencillos, tomar datos de todas las funciones MPI.

```
> mpicc -mpe=mpilog -o prog prog.c
> mpiexec –n xx prog
```

Una vez compilado, lo ejecutamos y obtenemos un fichero de trazas, de tipo `clog2`. Ahora podemos ejecutar:

```
> jumpshot
```

para analizar el fichero de trazas obtenido. `Jumphot4` trabaja con un formato de tipo `slog2`, por lo que previamente hay que efectuar una conversión a dicho formato. Esa conversión puede hacerse ya dentro de la aplicación, o ejecutando:

```
> clog2TOslog2 prog.clog2
```

La ventana inicial de `jumpshot` nos permite escoger el fichero que queremos visualizar. En una nueva ventana aparece un esquema gráfico de la ejecución, en el que las diferentes funciones de MPI se representan con diferentes colores. En el caso de las comunicaciones punto a punto, una flecha une la emisión y recepción de cada mensaje.

Con el botón derecho del ratón podemos obtener datos de las funciones ejecutadas y de los mensajes transmitidos.

NOTA: para poder ejecutar esta aplicación gráfica (u otras) de manera remota:

-- desde windows: ejecutar previamente `X-Win32` (o una aplicación similar)
-- desde linux: entrar en la máquina ejecutando `ssh –X -l cuenta máquina`

```
/******************************************************************************
        hola.c        programa MPI: activacion de procesos
******************************************************************************/

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

int main(int argc, char *argv[])
{
  int   lnom;
  char  nombrepr[MPI_MAX_PROCESSOR_NAME];cat circu
  int   pid, npr;
  int   A = 2;                          // identificador y numero de proc.

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &npr);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);

  MPI_Get_processor_name(nombrepr, &lnom);

  A = A + 1;
  printf(" >> Proceso %2d de %2d activado en %s, A = %d\n", pid, npr, nombrepr, A);

  MPI_Finalize();
  return 0;
}
```

```
/******************************************************************************
        circu.c        paralelizacion MPI de un bucle
******************************************************************************/

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

#define DECBIN(n,i) ((n&(1<<i))?1:0)

void test (int pid, int z)
{
  int v[16], i;

  for (i=0; i<16; i++) v[i] = DECBIN(z,i);

  if ((v[0] || v[1]) && (!v[1] || !v[3]) && (v[2] || v[3])
     && (!v[3] || !v[4]) && (v[4] || !v[5])
     && (v[5] || !v[6]) && (v[5] || v[6])
     && (v[6] || !v[15]) && (v[7] || !v[8])
     && (!v[7] || !v[13]) && (v[8] || v[9])
     && (v[8] || !v[9]) && (!v[9] || !v[10])
     && (v[9] || v[11]) && (v[10] || v[11])
     && (v[12] || v[13]) && (v[13] || !v[14])
     && (v[14] || v[15]))
  {
    printf(" %d) %d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d (%d)\n", pid, v[15],v[14],v[13],
      v[12],v[11],v[10],v[9],v[8],v[7],v[6],v[5],v[4],v[3],v[2],v[1],v[0], z);
    fflush(stdout);
  }
}

int main (int argc, char *argv[])
{
  int   i, pid, npr;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &npr);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);

  for (i=pid; i<65536; i += npr)  test(pid, i);

  MPI_Finalize();
  return 0;
}
```

```
/*****************************************************************************
        envio.c
         se envia un vector desde el procesador 0 al 1
*****************************************************************************/


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

#define N 10


int main (int argc, char **argv)
{
  int   pid, npr, origen, destino, ndat, tag;
  int   VA[N], i;
  MPI_Status    info;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  MPI_Comm_size(MPI_COMM_WORLD, &npr);

  for (i=0; i<N; i++) VA[i] = 0;

  if (pid == 0)
  {
    for (i=0; i<N; i++) VA[i] = i;

    destino = 1; tag = 0;
    MPI_Send(&VA[0], N, MPI_INT, destino, tag, MPI_COMM_WORLD);
  }

  else if (pid == 1)
  {
    printf("\nvalor de VA en P1 antes de recibir datos\n\n");
    for (i=0; i<N; i++) printf("%4d", VA[i]);
    printf("\n\n");

    origen = 0; tag = 0;
    MPI_Recv(&VA[0], N, MPI_INT, origen, tag, MPI_COMM_WORLD, &info);

    MPI_Get_count (&info, MPI_INT, &ndat);
    printf("Pr 1 recibe VA de Pr %d: tag %d, ndat %d \n\n",
            info.MPI_SOURCE, info.MPI_TAG, ndat);
    for (i=0; i<ndat; i++) printf("%4d", VA[i]);
    printf("\n\n");
  }

  MPI_Finalize();
  return 0;
}
```

EJERCICIOS

>> **envio1** Modifica el programa `envio` para que devuelva a P0 la suma de los elementos del vector recibido y éste último imprima el resultado.

>> **circu1** Modifica el programa `circu` para que P0 imprima el número de soluciones obtenidas.

```
/*****************************************************************************
        dlock1.c
        intercambio de dos variables
*****************************************************************************/


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


int main (int argc, char **argv)
{
  int   pid, origen, destino, tag;
  int   A, B, C;
  MPI_Status    info;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);

  if (pid == 0)
  {
    A = 5;

    printf("\n    >> recibiendo datos de P1  \n");
    origen = 1;  tag = 1;
    MPI_Recv(&B, 1, MPI_INT, origen, tag, MPI_COMM_WORLD, &info);

    printf("\n    >> enviando datos a P1  \n");
    destino = 1; tag = 0;
    MPI_Send(&A, 1, MPI_INT, destino, tag, MPI_COMM_WORLD);

    C = A + B;
    printf("\n   C es %d en proc 0 \n\n", C);
  }
  else if (pid == 1)
  {
    B = 6;

    printf("\n    >> recibiendo datos de P0  \n");
    origen = 0;  tag = 1;
    MPI_Recv(&A, 1, MPI_INT, origen, tag, MPI_COMM_WORLD, &info);

    printf("\n    >> enviando datos a P0  \n");
    destino = 0; tag = 0;
    MPI_Send(&B, 1, MPI_INT, destino, tag, MPI_COMM_WORLD);

    C = A + B;
    printf("\n   C es %d en proc 1 \n\n", C);
  }

  MPI_Finalize();
  return 0;
}
```

```
/*****************************************************************************
       send-dead.c
       prueba para ver el tamaino del buffer de la funcion send
       el programa se bloquea al enviar un paquete mas grande
*****************************************************************************/

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

#define N 100000
int main(int argc, char** argv)
{
  int         pid, kont;                // Identificador del proceso
  int         a[N], b[N], c[N], d[N];
  MPI_Status  status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);

  for (kont=100; kont<=N; kont=kont+100)
  {
    if (pid == 0)
    {
      MPI_Send(&a[0], kont, MPI_INT, 1, 0, MPI_COMM_WORLD);
      MPI_Recv(&b[0], kont, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
      printf("emisor %d \n", kont);
    }
    else
    {
      MPI_Send(&c[0], kont, MPI_INT, 0, 0, MPI_COMM_WORLD);
      MPI_Recv(&d[0], kont, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
      printf("                   receptor %d \n", kont);
     }
  }

  MPI_Finalize();
  return 0;
} /*  main  */
```

```
/*****************************************************************************
       ssend.c
       Ping-pong entre dos procesadores
       Comunicacion sincrona ssend
*****************************************************************************/

#include <stdio.h>
#include <mpi.h>
#include <math.h>
#include <unistd.h>

#define VUELTAS 4

double calculo()
{
  double aux;

  sleep(1);
  aux = rand() % 100;

  return(aux);
}


int main(int argc, char** argv)
{
  double      t0, t1, dat= 0.0, dat1, dat_rec;
  int         pid, i;
  MPI_Status  status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);

  if (pid == 0) t0 = MPI_Wtime();

  for(i=0; i<VUELTAS; i++)
  {
    if (pid == 0)
    {
      MPI_Ssend(&dat, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);

      dat1 = calculo();

      MPI_Recv(&dat_rec, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &status);
      dat = dat1 + dat_rec;
    }
    else
    {
      dat1 = calculo();

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

      dat = dat1 + dat_rec;
      MPI_Ssend(&dat, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }
  }

  if (pid == 0)
  {
    t1 = MPI_Wtime();
    printf("\n   Tiempo de ejecucion = %f s \n", t1-t0);
    printf("\n   Dat = %1.3f \n\n", dat);
  }

  MPI_Finalize();
  return 0;
} /*  main  */
```

```
/*****************************************************************************
       isend.c
       Ping-pong entre dos procesadores
       Comunicacion inmediata isend
*****************************************************************************/

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

#define VUELTAS 4

double calculo()
{
  double aux;

  sleep(1);
  aux = rand() % 100;

  return(aux);
}

int main(int argc, char** argv)
{
  double      t0, t1, dat = 0.0, dat1, dat_rec;
  int         pid, i;
  MPI_Status  status;
  MPI_Request request;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);

  if (pid == 0) t0 = MPI_Wtime();

  for(i=0; i<VUELTAS; i++)
  {
    if (pid == 0)
    {
      MPI_Isend(&dat, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD,&request);

      dat1 = calculo();

      MPI_Wait(&request, &status);
      MPI_Recv(&dat_rec, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &status);
      dat = dat1 + dat_rec;
    }
    else
    {
      dat1 = calculo();

      if (i!=0) MPI_Wait(&request, &status);
      MPI_Recv(&dat_rec, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);

      dat = dat1 + dat_rec;
      MPI_Isend(&dat, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,&request);
    }
  }

  if (pid == 0)
  {
    t1 = MPI_Wtime();
    printf("\n   Tiempo de ejecucion = %f s \n", t1-t0);
    printf("\n   Dat = %1.3f \n\n", dat);
  }

  MPI_Finalize();
  return 0;
} /*  main  */
```

```
/************************************************************
     probe.c
     ejemplo de uso de la funcion probe de MPI
************************************************************/

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


int main (int argc, char **argv)
{
  int   pid, npr, origen, destino, ndat, tag;
  int   i, longitud, tam;
  int   *VA, *VB;
  MPI_Status    info;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  MPI_Comm_size(MPI_COMM_WORLD, &npr);

  if (pid == 0)
  {
    srand(time(NULL));
    longitud = rand() % 100;
    VA = (int *) malloc (longitud*sizeof(int));
    for (i=0; i<longitud; i++) VA[i] = i;

    printf("\n valor de VA en P0 antes de enviar los datos\n\n");
    for (i=0; i<longitud; i++) printf("%4d", VA[i]);
    printf("\n\n");

    destino = 1; tag = 0;
    MPI_Send(VA, longitud, MPI_INT, destino, tag, MPI_COMM_WORLD);
  }

  else if (pid == 1)
  {
    origen = 0;  tag = 0;
    MPI_Probe(origen, tag, MPI_COMM_WORLD, &info);
    MPI_Get_count(&info, MPI_INT, &tam);

    if(tam != MPI_UNDEFINED)
    {
      VB = (int *) malloc(tam*sizeof(int));
      MPI_Recv(VB, tam, MPI_INT, origen, tag, MPI_COMM_WORLD, &info);
    }

    printf("\n valor de VB en P1 tras recibir los datos\n\n");
    for (i=0; i<tam; i++) printf("%4d", VB[i]);
    printf("\n\n");
  }

  MPI_Finalize();
  return 0;
}
```

EJERCICIO

>> Completa el programa `intepar0`, que calcula la integral de la funcion f = 1/(x+1) + 1/(x^2+1) mediante sumas de areas de trapecios bajo la curva que representa la función.

```
/*****************************************************************************
       intepar0.c
       Integral de una funcion mediante sumas de areas de trapecios
        -- para completar
******************************************************************************/

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

double   t0, t1;
void   Leer_datos(double* a_ptr, double* b_ptr, int* n_ptr, int pid, int npr);
double Integrar(double a_loc, double b_loc, int n_loc, double w);
double f(double x);


int main(int argc, char** argv)
{
  int         pid, npr;                 // Identificador y numero de proc.
  double      a, b, w, a_loc, b_loc;
  int         n, n_loc, resto, kont;
  double      resul, resul_loc;         // Resultado de la integral

  int         origen, destino, tag;
  MPI_Status  status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  MPI_Comm_size(MPI_COMM_WORLD, &npr);

// Lectura y distribucion de parametros a todos los procesadores
  Leer_datos(&a, &b, &n, pid, npr);

  w = (b-a) / n;        // cada uno calcula el trozo que tiene que sumar
  n_loc = n / npr;
  resto = n % npr;
  if (pid < resto) n_loc = n_loc + 1;
  a_loc = a + pid * n_loc * w;
  if (pid >= resto) a_loc = a_loc + resto * w;
  b_loc = a_loc + n_loc * w;


// Calculo de la integral local
  resul_loc = Integrar(a_loc, b_loc, n_loc, w);

// Sumar resultados parciales
//  (POR COMPLETAR)

// Impresion de resultados
  if (pid == 0)
  {
    t1 = MPI_Wtime();
    printf("\n Integral (= ln x+1 + atan x), de %1.1f a %1.1f, %d trap. = %1.12f\n", a,b,n,resul);
    printf("    Tiempo ejecucion (%d pr.) = %1.3f ms \n\n", npr, (t1-t0)*1000);
  }

  MPI_Finalize();
  return 0;
} /*  main  */
```

```
// FUNCION Leer_datos
void Leer_datos(double* a_ptr, double* b_ptr, int* n_ptr, int pid, int npr)
{
  int           origen, destino, tag;
  float         aux_a, aux_b;
  MPI_Status    status;

  if (pid == 0)
  {
    printf("\n  Introduce a, b (limites) y n (num. de trap.<= 10e8)  \n");
    scanf("%f %f %d", &aux_a, &aux_b, n_ptr);

    t0 = MPI_Wtime();

    (*a_ptr) = (double)aux_a;    // a_ptr, b_ptr, n_ptr: punteros a a, b y n
    (*b_ptr) = (double)aux_b;


// distribuir datos a todos los procesos
//   (POR COMPLETAR)

  }
} /* Leer_datos */



// FUNCION Integrar: calculo local de la integral
double Integrar(double a_loc, double b_loc, int n_loc, double h)
{
  double resul_loc, x;
  int    i;

// calculo de la integral
  resul_loc = (f(a_loc) + f(b_loc)) / 2.0;
  x = a_loc;

  for (i=1; i<n_loc; i++)
  {
    x = x + w;
    resul_loc = resul_loc + f(x);
  }
  resul_loc = resul_loc * w;

  return resul_loc;
} /*  Integrar  */


// FUNCION f: funcion a integrar
double f(double x)
{
  double y;
  y = 1.0 / (x + 1.0) + 1.0 / (x*x + 1.0) ;
  return y;
}
```

```
/*****************************************************************************
       colec.c
       Pruebas de comunicaciones colectivas
       ejecutar con 4 procesos
*****************************************************************************/

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

int main(int argc, char** argv)
{
  int pid, npr;
  int i, X;
  int A[8], B[2], C[2];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  MPI_Comm_size(MPI_COMM_WORLD, &npr);

/*   Inicializaciones en todos los procesos */
  for(i=0; i<8; i++) A[i] = -1;
  for(i=0; i<2; i++) B[i] = -1;
  for(i=0; i<2; i++) C[i] = -1;
  X = -1;

  if (pid == 0)
  {  for(i=0; i<8; i++) A[i] = i;
     X = 6;
  }

/* Broadcast de X desde P0 */
  MPI_Bcast(&X, 1, MPI_INT, 0, MPI_COMM_WORLD);
  printf("1: BC de P0  --> P%d y X = %d \n", pid, X);

  MPI_Barrier(MPI_COMM_WORLD);

/* Reparto de A (8 elementos) desde P0 en trozos de tamaino 2, en B */
  MPI_Scatter(&A[0], 2, MPI_INT, &B[0], 2, MPI_INT, 0, MPI_COMM_WORLD);
  printf(" 2: SCATTER  soy P%d y B = %d %d \n", pid, B[0], B[1]);

  MPI_Barrier(MPI_COMM_WORLD);

/* Suma de los 4 vectores B (2 elementos) en el vector C en P0 */
  MPI_Reduce(&B[0], &C[0], 2, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
  printf("  3: REDUCE en P%d  C = %d %d \n", pid, C[0], C[1]);

  MPI_Barrier(MPI_COMM_WORLD);

/* Suma de los 4 vectores B (2 elementos) en el vector C en todos los P */
  MPI_Allreduce(&B[0], &C[0], 2, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
  printf("   4. ALLREDUCE   en P%d  C = %d %d \n", pid, C[0], C[1]);

  MPI_Barrier(MPI_COMM_WORLD);

/* Recoleccion de los vectores B en el vector A en P0 */
  for(i=0; i<2; i++) B[i] = B[i] + 10;
  MPI_Gather(&B[0], 2, MPI_INT, &A[0], 2, MPI_INT, 0, MPI_COMM_WORLD);
  printf("       5:  GATHER  en  P%d,  A  =  %d  %d  %d  %d  %d  %d  %d  %d\n",  pid,
A[0],A[1],A[2],A[3],A[4],A[5],A[6],A[7]);

  MPI_Barrier(MPI_COMM_WORLD);

/* Recoleccion de los vectores B en el vector A en todos los P */
  MPI_Allgather(&B[0], 2, MPI_INT, &A[0], 2, MPI_INT, MPI_COMM_WORLD);
  printf("        6:  ALLGATHER  en  P%d,  A  =  %d  %d  %d  %d  %d  %d  %d  %d\n",  pid,
A[0],A[1],A[2],A[3],A[4],A[5],A[6],A[7]);

  MPI_Finalize();
  return 0;
} /*  main  */
```

```
/*****************************************************************************
       BCsinc.c
       prueba de sincronicidad del BC
*****************************************************************************/

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

#define N 100

int main(int argc, char** argv)
{
  int pid, A[N];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);

  sleep(pid*2);

  printf("\n >> P%d: comienzo del BC \n", pid);

  MPI_Bcast(A, N, MPI_INT, 0, MPI_COMM_WORLD);

  printf("\n                          fin del BC en P%d \n", pid);

  MPI_Finalize();
  return 0;
} /*  main  */
```

```
/*****************************************************************************
     scatterv.c ejemplo del uso de la funcion scatterv
     Reparto de un vector de 8 elementos entre 4 proces.: 2, 4, 1 y 1 elementos
     Solo para el caso particular de 4 procesos
*****************************************************************************/

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

int main(int argc, char** argv)
{
  int pid, npr, i;
  int A[8], B[8], tam[4], desp[4];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  MPI_Comm_size(MPI_COMM_WORLD, &npr);

/*   Inicializaciones en todos los procesos */
  for(i=0; i<8; i++) { A[i] = -1; B[i] = -1; }
  if (pid == 0) {
     for(i=0; i<8; i++) A[i] = i;
     printf("\n A en P0 = %2d %2d %2d %2d %2d %2d %2d %2d\n",A[0],A[1],A[2],A[3],A[4],A[5],A[6],A[7]);
  }

/* Reparto de A (8 elementos) desde P0 en 4 trozos: 2, 4, 1, 1 elementos */
/* vector de tamainos de los trozos */
  tam[0] = 2;   tam[1] = 4;   tam[2] = tam[3] = 1;

/* vector de desplazamientos desde el origen a cada trozo */
  desp[0] = 0;   desp[1] = 2;   desp[2] = 6;   desp[3] = 7;

  MPI_Scatterv(&A[0], tam, desp, MPI_INT, &B[0], tam[pid], MPI_INT, 0, MPI_COMM_WORLD);

  printf("\n SCATTERV: P%d B =%2d %2d %2d %2d %2d %2d %2d %2d\n",pid,B[0],B[1], B[2],B[3],B[4],B[5],B[6],B[7]);

  MPI_Finalize();
  return 0;
} /*  main  */
```

EJERCICIO

>> Escribe y ejecuta una versión paralela del siguiente programa serie

```
/******************************************************************************
       sumvecser.c
       Calcula V(i) = V(i) * sumatorio(V(j))
       Codigo serie;
******************************************************************************/

#include <stdio.h>

int main(int argc, char **argv)
{
  int    N, i;
  int   *V, sum=0;

  printf("\nLongitud del vector: ");
  scanf("%d", &N);                      // Multiplo del numero de procesos

  V = (int *)malloc(sizeof(int) * N);
  for(i=0; i<N; i++) V[i] = rand() % 100 - 50;  // inicializacion del vector


// Procesamiento del vector

  for (i=0; i<N; i++) sum = sum + V[i];
  for (i=0; i<N; i++) V[i] = V[i] * sum;


// Imprimir unos resultados
  printf("\n    sum = %d\n", sum);
  printf("\n    V[0] = %d, V[N/2] = %d, V[N-1] = %d\n\n\n", V[0],V[N/2],V[N-1]);

  return 0;
}
```

```
/******************************************************************************
       intepar3.c      // FUNCION Leer_datos:   pack / unpack y broadcast
******************************************************************************/

void Leer_datos(double* a_ptr, double* b_ptr, int* n_ptr, int pid, int npr)
{
  float         aux_a, aux_b;
  char          buf[100];
  int           pos, root=0;
  if (pid == 0){
    ...  // lectura de los parametrosde la integral
    pos = 0;
    MPI_Pack(a_ptr, 1, MPI_DOUBLE, buf, 100, &pos, MPI_COMM_WORLD);
    MPI_Pack(b_ptr, 1, MPI_DOUBLE, buf, 100, &pos, MPI_COMM_WORLD);
    MPI_Pack(n_ptr, 1, MPI_INT, buf, 100, &pos, MPI_COMM_WORLD);
    MPI_Bcast(buf, pos, MPI_PACKED, root, MPI_COMM_WORLD);
  } else {
    MPI_Bcast(buf, 100, MPI_PACKED, root, MPI_COMM_WORLD);
    pos = 0;
    MPI_Unpack(buf, 100, &pos, a_ptr, 1, MPI_DOUBLE, MPI_COMM_WORLD);
    MPI_Unpack(buf, 100, &pos, b_ptr, 1, MPI_DOUBLE, MPI_COMM_WORLD);
    MPI_Unpack(buf, 100, &pos, n_ptr, 1, MPI_INT, MPI_COMM_WORLD);
  }
} /* Leer_datos */
```

EJERCICIO: mbloq.c

>> Escribe una versión paralela del programa `mbloq.c`, que calcula la suma módulo 10 de los elementos de una matriz `A`.
El proceso `P0` inicializa y reparte la matriz `A(NxN)` por bloques bidimensionales a `npr` procesos. Éstos efectúan la suma (módulo 10) de los elementos del correspondiente bloque y devuelven el resultado a P0.

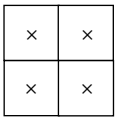
P0 envía a cada uno de los *workers*, uno a uno, la correspondiente submatriz. En cada mensaje se debe enviar una submatriz completa, para lo que hay que utilizar el **tipo de datos derivado "vector"**.
Ejecuta el programa con 1+4 (5), 1+9 (10), 1+16 (17) procesos.

(función para la raíz cuadrada: `sqrt();` añadir `#include <math.h>`)

```
/*****************************************************************************
        mbloqser.c
      -- ejercicio para paralelizar
      -- distribuir la matriz por bloques, definir tipos
*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>

#define N 180       // multiplo de 12 (2x2, 3x3, 4x4)

int main(int argc, char** argv)
{
  int    i, j;
  int    A[N][N];
  int    sum=0;

/* inicilaizacion de la matriz */
 for(i=0; i<N; i++)
  for(j=0; j<N; j++) A[i][j] = rand() % 20;

/* operación a paralelizar, distribuyendo la matriz por bloques */
  for(i=0; i<N; i++)
    for(j=0; j<N; j++) sum = (sum + A[i][j]) % 10;

  printf("\n   suma de A  % 10 =  %d\n\n", sum);
  return 0;
} /*  main  */
```

EJERCICIO: cuatro.c

>> Dos procesos paralelos inicializan las matrices A y B (N×N) respectivamente (ver código) e intercambian los cuatro valores centrales de cada cuarto de matriz (por bloques).
Por ejemplo, si N = 10, se trata de intercambiar los valores de las posiciones [2][2], [2][7], [7][2] y [7][7].



Sólo se debe utilizar un mensaje para enviar los cuatro datos
(a) utilizando el **tipo de datos indexed**
(b) utilizando **datos empaquetados**

```
/*****************************************************************************
      cuatro.c
      Intercambio entre P0 (A) y P1 (B) de los cuatro elementos centrales de los
      cuartos de cada matriz.
      -- por completar
*****************************************************************************/

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

#define N 10            // N par pero no multiplo de 4

int main(int argc, char** argv)
{
  int    pid, npr;
  int    i, j;
  int    A[N][N], B[N][N];
  MPI_Status  info;
  MPI_Datatype CUATRO;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  MPI_Comm_size(MPI_COMM_WORLD, &npr);

  ...

  if (pid == 0)
  {
    for(i=0; i<N; i++)
    for(j=0; j<N; j++) A[i][j] = i*N + j;

    ...

     printf("\n <Cuatro> de A(+) en P0:%d %d %d %d\n", A[N/4][N/4], A[N/4][3*N/4],
A[3*N/4][N/4], A[3*N/4][3*N/4]);
  }
  else if (pid == 1)
  {
    for(i=0; i<N; i++)
    for(j=0; j<N; j++) B[i][j] = -(i*N + j);

    ...

    printf("\n <Cuatro> de B(-) en P1: %d %d %d %d\n", B[N/4][N/4], B[N/4][3*N/4],
B[3*N/4][N/4], B[3*N/4][3*N/4]);
  }
  MPI_Finalize();
  return 0;
} /*  main  */
```

```
/*****************************************************************************
   topol_malla.c
   Creacion de comunicadores en malla, filas y columnas
   Salida:      coordenadas de los nodos en una topologia 2D;
                BC en filas y columnas de esa topologia
     1.  Crea un comunicador en malla 2D (toro) a partir del global
     2.  Imprime las coordenadas de los nodos en la nueva topologia
     3.  Usa MPI_Cart_sub  para crear comunicadores para las filas
     4.  Efectua un BC en cada fila e imprime los resultados
     3.  Usa MPI_Cart_sub  para crear comunicadores para las columnas
     4.  Efectua un BC en cada columna e imprime los resultados
    Nota: El numero de procesadores debe ser un cuadrado perfecto k x k
*****************************************************************************/

#include <stdio.h>
#include "mpi.h"
#include <math.h>

int main(int argc, char* argv[])
{
  int       pid, pid_malla, pid_m, pid_fila, pid_col;
  int       npr, k, longi_dim[2], anillo[2], reordenar = 1;
  int       coord[2], coord_libre[2], pruebaf, pruebac;
  MPI_Comm  malla_com, fila_com, col_com;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &npr);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);

// datos de la topologia en malla
  k = sqrt(npr);
  longi_dim[0] = longi_dim[1] = k;      // numero de nodos en cada dimension
  anillo[0] = anillo[1] = 1;            // anillo (1) o cadena (0) en cada dim.

// creacion del nuevo comunicador con una malla asociada
  MPI_Cart_create(MPI_COMM_WORLD, 2, longi_dim, anillo, reordenar, &malla_com);

//coordenadas del procesador en la nueva topologia en malla
  MPI_Comm_rank(malla_com, &pid_malla);         //pid en la malla (reord=1!)
  MPI_Cart_coords(malla_com, pid_malla, 2, coord);

//a la inversa, partiendo de las coordenadas, devuelve el valor del pid en la malla
  MPI_Cart_rank(malla_com, coord, &pid_m);

  if (pid==0) printf("\n ---- coordenadas en la malla \n");
  printf("Proceso %d > pid_malla = %d, coords = (%d,%d), pid_m = %d\n",
        pid, pid_malla, coord[0], coord[1], pid_m);

//creacion de comunicadores para las filas de la malla
  coord_libre[0] = 0;
  coord_libre[1] = 1;
  MPI_Cart_sub(malla_com, coord_libre, &fila_com);

// prueba de comunicacion: el primer nodo de cada fila envia su pid al resto
  if (coord[1] == 0) pruebaf = pid;
  MPI_Bcast(&pruebaf, 1, MPI_INT, 0, fila_com);

  MPI_Comm_rank(fila_com, &pid_fila);
  if (pid==0) printf("\n ---- prueba de BC por filas \n");
  printf("Proceso %d > coords = (%d,%d), pid_primero_fila = %d, pid_fila: %d\n",
         pid, coord[0], coord[1], pruebaf, pid_fila);

//creacion de comunicadores para las columnas de la malla
  coord_libre[0] = 1;
  coord_libre[1] = 0;
  MPI_Cart_sub(malla_com, coord_libre, &col_com);

// prueba de comunicacion: el primer nodo de cada fila envia su pid al resto
  if (coord[0] == 0) pruebac = pid;
  MPI_Bcast(&pruebac, 1, MPI_INT, 0, col_com);

  MPI_Comm_rank(col_com, &pid_col);
  if (pid==0) printf("\n ---- prueba de BC por columnas \n");
  printf("Proceso %d > coords = (%d,%d), pid_primero_col = %d, pid_col: %d\n",
         pid, coord[0], coord[1], pruebac, pid_col);

  MPI_Finalize();
  return 0;
}  /* main */
```

EJERCICIO: matvec.c

>> Escribe la versión paralela del programa `matvecser`, que efectúa el siguiente cálculo con matrices y vectores.

```
double A(NxN), B(N), C(N), D(N), PE
C(N) = A(NxN) × B(N)
D(N) = A(NxN) × C(N)
PE = sum (C(N) . D(N))
```

El programa pide al principio el tamaño de los vectores, `N`.
Inicialmente, `A` y `B` están en el proceso `P0;` al final, `C`, `D` y `PE` tienen que quedar en `P0`.

```
/*****************************************************************************
      matvecser.c
*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  int    N, i, j;
  double PE = 0.0;
  double *Am, **A, *B, *C, *D;

  printf("\n Longitud de los vectores (<1000) = ");
  scanf("%d" , &N);

/* reserva de memoria */
  Am = (double *) malloc (N*N * sizeof(double));
  A = (double **) malloc (N * sizeof(double *));    // para usar dos dimensiones
  for(i=0; i<N; i++) A[i] = &Am[i*N];

  B = (double *) malloc (N * sizeof(double));
  C = (double *) malloc (N * sizeof(double));
  D = (double *) malloc (N * sizeof(double));

/* inicializar variables */
  for(i=0; i<N; i++)
  {
    for(j=0; j<N; j++) A[i][j] = (double) (N-i)*0.1/N;
    B[i] = (double) i*0.05/N;
  }

/* comienzo del calculo */
  for (i=0; i<N; i++)
  {
    C[i] = 0.0;
    for (j=0; j<N; j++) C[i] = C[i] + Am[i*N+j] * B[j];    // o A[i][j]
  }

  PE = 0.0;
  for (i=0; i<N; i++)
  {
    D[i] = 0.0;
    for (j=0; j<N; j++) D[i] = D[i] + Am[i*N+j] * C[j];
    PE = PE + D[i]*C[i];
  }

/* resultados */
  printf("\n\n   PE = %1.3f \n", PE);
  printf("  D[0] = %1.3f, D[N/2] = %1.3f, D[N-1)] = %1.3f\n\n", D[0],D[N/2],D[N-1]);
  return 0;
}
```

```
/*************************************************************************
        iopar1.c
        P0 escribe en el fichero dat1 4 enteros por cada proceso
        Luego todos leen los 4 enteros que les corresponden
*************************************************************************/

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

#define tambuf 4*32

int main(int argc, char **argv)
{
  int    pid, npr;
  int    i, numdat;
  int    buf[tambuf], buf2[tambuf], modoacceso;

  MPI_File  dat1;
  MPI_Status info;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  MPI_Comm_size(MPI_COMM_WORLD, &npr);

  numdat = 4;
  if (pid == 0) for(i=0; i<npr*numdat; i++) buf[i] = i*i;

  if (pid == 0)
  {
    modoacceso = (MPI_MODE_CREATE | MPI_MODE_WRONLY);
    MPI_File_open(MPI_COMM_SELF, "dat1", modoacceso, MPI_INFO_NULL, &dat1);
    MPI_File_seek(dat1, 0, MPI_SEEK_SET);
    MPI_File_write(dat1, buf, npr*numdat, MPI_INT, &info);
    MPI_File_close(&dat1);

    printf("\n P0 ha escrito %d datos, desde 0 hasta %d \n\n", npr*numdat, buf[npr*numdat-1]);
  }

  sleep(3);

  modoacceso = MPI_MODE_RDONLY;

  MPI_File_open(MPI_COMM_WORLD, "dat1", modoacceso, MPI_INFO_NULL, &dat1);
  MPI_File_seek(dat1, pid*numdat*sizeof(int), MPI_SEEK_SET);
  MPI_File_read(dat1, buf2, numdat, MPI_INT, &info);
  MPI_File_close(&dat1);

  printf("  > P%d ha leido %5d %5d %5d %5d \n", pid, buf2[0], buf2[1], buf2[2], buf2[3]);

  MPI_Finalize();
  return 0;
}
```

```
/*****************************************************************************
        iopar2.c

  Cada proceso escribe en el fichero dat1 4 enteros, cada uno a continuacion del siguiente
  Luego, cada proceso lee los 4 entreos escritos por el proceso pid+1
*****************************************************************************/

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

#define tambuf 4*32

int main(int argc, char **argv)
{
  int    pid, npr;
  int    i, numdat, numint;
  int    buf[tambuf], buf2[tambuf], modoacceso;

  MPI_File  dat1;
  MPI_Status info;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  MPI_Comm_size(MPI_COMM_WORLD, &npr);

  numdat = 4;
  numint = numdat * sizeof(int);
  for(i=0; i<numdat; i++) buf[i] = (i+pid*numdat)*(i+pid*numdat);


  modoacceso =  (MPI_MODE_CREATE | MPI_MODE_RDWR);
  MPI_File_open(MPI_COMM_SELF, "dat1", modoacceso, MPI_INFO_NULL, &dat1);
  MPI_File_seek(dat1, pid*numint, MPI_SEEK_SET);
  MPI_File_write(dat1, buf, numdat, MPI_INT, &info);
  MPI_File_sync(dat1);

  MPI_Barrier(MPI_COMM_WORLD);
  sleep(3);

  MPI_File_seek(dat1, ((pid+1)%npr)*numint, MPI_SEEK_SET);
  MPI_File_read(dat1, buf2, numdat, MPI_INT, &info);
  MPI_File_close(&dat1);

  printf("\n > Pid %d ha leido %5d %5d %5d %5d \n", pid, buf2[0], buf2[1], buf2[2], buf2[3]);
  MPI_Finalize();
  return 0;
}
```

```
/**************************************************************************
        iopar3.c

        P0 escribe en un fichero en formato ascii una matriz de 8x8 enteros
        (es una prueba: el resto no hace nada)
**************************************************************************/

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

#define tambuf 100
#define N 64

int main(int argc, char **argv)
{
  int    pid, npr;
  int    i;
  int    buf[tambuf], buf2[tambuf], modoacceso;

  char aux[20];
  char linea[2000];

  MPI_File  dat2;
  MPI_Status info;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  MPI_Comm_size(MPI_COMM_WORLD, &npr);


  if (pid == 0) for(i=0; i<N; i++) buf[i] = i*i;

// Los 64 elementos de buf se convierten a ascci, 8 lineas de 8 elementos

  linea[0] = '\0';
  for(i=0; i<N; i++)
  {
    if((i%8==0) && (i>0)) strcat(linea,"\n");
    sprintf (aux, " %6d", buf[i]);
    strcat (linea, aux);
  }
  strcat(linea,"\n");

// ahora se escribe linea

  if (pid == 0)
  {
    modoacceso = (MPI_MODE_CREATE | MPI_MODE_WRONLY);
    MPI_File_open(MPI_COMM_SELF, "dat2", modoacceso, MPI_INFO_NULL, &dat2);
    MPI_File_seek(dat2, 0, MPI_SEEK_SET);
    MPI_File_write(dat2, linea, strlen(linea), MPI_CHAR, &info);
    MPI_File_close(&dat2);
  }
  MPI_Finalize();
  return 0;
}
```

```
/*************************************************************************
         iopar4.c

        Leen entre todos los procesos  el fichero ascci creado por iopar3.c
        y lo convierten a enteros. El fichero es de 8 lineas de 8 enteros.
        (ejecutar con 2, 4 u 8 procesos)
*************************************************************************/


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

#define N  64
#define M  8*(8*7+1)    // numero total de caracteres del fichero dat2

int main(int argc, char **argv)
{
  int    pid, npr;
  int    i, k;
  int    buf[N], modoacceso;
  char   aux[M];

  MPI_File  dat2;
  MPI_Status info;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  MPI_Comm_size(MPI_COMM_WORLD, &npr);


  modoacceso =  MPI_MODE_RDONLY;

  MPI_File_open(MPI_COMM_WORLD, "dat2", modoacceso, MPI_INFO_NULL, &dat2);
  MPI_File_seek(dat2, pid*M/npr, MPI_SEEK_SET);
  MPI_File_read(dat2, aux, M/npr, MPI_CHAR, &info);
  MPI_File_close(&dat2);

// convertimos de ascci a enteros
  k = 0;
  for(i=0; i<N/npr; i++)
  {
    sscanf(&aux[k], "%d", &buf[i]);
    k = k+7;
   if ((i%8==0) && (i>0)) k++;
  }

// cada proceso imprime los dos primeros y los dos ultimos datos que ha leido
  printf("\n  P%d  ha  leido     %5d  %5d  ...  %5d  %5d  \n",  pid,  buf[0],  buf[1],
buf[N/npr-2], buf[N/npr-1]);

  MPI_Finalize();
  return 0;
}
```

EJERCICIO

>> Completa el programa `iopar5.c`

P0 escribe en el fichero fmax 10000 enteros random entre 0 y 313

Luego todos los procesos buscan en paralelo la diferencia maxima entre dos elementos consecutivos, y su posicion.

```
/*************************************************************************
        iopar5.c
*************************************************************************/

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

#define N 10000

int main(int argc, char **argv)
{
  int    pid, npr, i;
  int    buf[N], buf2[N];
  int    dmax = 0, dmaxl = 0, pmax, pmaxl;
  int    modoacceso, numdat, ini;

  MPI_File  fmax;
  MPI_Status info;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
  MPI_Comm_size(MPI_COMM_WORLD, &npr);


  if (pid == 0)
  {
    // genera los datos
    for(i=0; i<N; i++) buf[i] = (int) rand() % 314;

    // prueba:  P0 hace los calculos
    for (i=0; i<N-1; i++)
      if ( (buf[i+1] - buf[i]) > dmax)
      {
         dmax = buf[i+1] - buf[i];
         pmax = i;
      }
    printf("\n TEST de PRUEBA:  dmax = %d  pmax = %d\n\n", dmax, pmax);
    dmax = 0; pmax = 0;


    // escribe los datos en el fichero
    //  PARA COMPLETAR

  }

  sleep(3);

  // Cada proceso va a leer un trozo consecutivo del fichero;
  // cada uno empieza en el ultimo dato del anterior
  // ini: comienzo de mi trozo; numdat: datos que me corresponde analizar
  //  ojo con la division entera!
  //   PARA COMPLETAR
```

```
  // calculo local

  for(i=0; i<(numdat-1); i++)
    if ( (buf2[i+1] - buf2[i]) > dmaxl )
    {
      dmaxl = buf2[i+1] - buf2[i];
      pmaxl = i + ini;
    }

  printf(" Result. locales en P%d: dmaxl = %d  pmaxl = %d \n", pid, dmaxl, pmaxl);

  // Calculo del maximo global y de su posicion (la primera vez que aparece en la
lista)
  // PARA COMPLETAR




  if (pid==0) printf("\n RESULTADO FINAL: dmax = %d  pmax = %d \n\n", dmax, pmax);

  MPI_Finalize();
  return 0;
}
```

```
/*****************************************************************************
        pproductor.c
        Para ejecutar con pconsumidor.c, con un modelo MPMD (pc1 o pc_schema)
*****************************************************************************/

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

#define DATOS   0  //tipos de tag
#define CONTROL 1

#define NFIL 20  //dimensiones de la matriz
#define NCOL 10

float matriz[NFIL][NCOL];

void producir_datos(int vez, int fila)
{
  int i;
  for(i=0; i<NCOL; i++) matriz[fila][i] = (float)vez + (float)fila;
  printf("PROD: vez(%d)-fila(%d)->[%.2f]\n", vez+1, fila, matriz[fila][0]);
}

int main(int argc, char** argv)  // productor
{
  int         pid;              // Identificador del proceso
  int         nveces, i, nfila;
  MPI_Status  status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);

  printf("Numero de veces a transmitir filas de la matriz[%d][%d]-->  ",NFIL,NCOL);
  scanf("%d", &nveces);

  for(i=0; i<nveces; i++)
  {
    for (nfila=0; nfila<NFIL; nfila++)
    {
      producir_datos(i,nfila);
      MPI_Ssend(&matriz[nfila][0], NCOL, MPI_FLOAT, 1, DATOS, MPI_COMM_WORLD);
    }
  }
  MPI_Ssend(&matriz[0][0], NCOL, MPI_FLOAT, 1, CONTROL, MPI_COMM_WORLD);
  printf ("\nFIN proceso PRODUCTOR con pid: %d\n\n", pid);

  MPI_Finalize();
  return 0;
} /*  main  */
```

```
/*****************************************************************************
        pconsumidor.c
        Para ejecutar con pproductor.c, con un modelo MPMD (pc1 o pc_schema)
*****************************************************************************/

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

#define DATOS   0  //tipos de tag
#define CONTROL 1

#define MAXIMA_LONG 10  //dimension buffer consumidor

void consumir_datos(float buffer[], int longitud)
{
  int i;
  float suma = 0.0;

  for(i=0; i<longitud; i++) suma += buffer[i];
  printf("\t\t\t\tCONSUM: buffer[]=%3.2f ->suma:%3.2f\n", buffer[0], suma );
}

int main(int argc, char** argv) //consumidor
{
  int         pid;              // Identificador del proceso
  int         nveces, i, nfila;
  MPI_Status  status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &pid);

  float buffer_consum[MAXIMA_LONG];
  int longitud;

  while (1)
   {
     MPI_Recv(&buffer_consum[0], MAXIMA_LONG, MPI_FLOAT, 0,
                            MPI_ANY_TAG, MPI_COMM_WORLD, &status);
     if (status.MPI_TAG == CONTROL) break;
     MPI_Get_count(&status, MPI_FLOAT, &longitud);
     consumir_datos(&buffer_consum[0], longitud);
   }
  printf ("\n\n\t\t\t\tFIN proceso CONSUMIDOR con pid: %d\n\n", pid);

  MPI_Finalize();
  return 0;
} /*  main  */
```

--- la version pprod_consum.c, todo junto

Programas para lanzar ambos programas

pc1

```
-----------
n0  productor
n1  consumidor
```

pc_schema

```
---------
c0  productor
c1  consumidor
```