GRADO EN INGENIERÍA DE TECNOLOGÍAS DE LA INFORMACIÓN

PROCESAMIENTO PARALELO

MPI: Ejemplos de Comunicadores y Grupos

David Moreno Salinas
Victorino Sanz Prat

MPI: Ejemplos de Comunicadores y Grupos

En este documento se mostrarán una serie de ejemplos para aclarar algunos conceptos claves dentro de la programación de aplicaciones paralelas usando el estándar MPI. Estos conceptos serán los comunicadores (tanto intra como intercomunicadores), sus universos de comunicación asociados, grupos de procesos y las funciones relativas a la manipulación de los mismos.

1. INTRACOMUNICADORES

Comenzaremos con la ejecución estática de procesos y los intracomunicadores. Para ello usaremos el siguiente código de ejemplo:

```
/*****************************************************************************
 * Fichero: Ejemplo1.c
 * Ejemplo de funcionamiento de primitivas de MPI y universos de comunicación
 * de los comunicadores
 * Compilar con:    mpicc Ejemplo1.c –o Ejemplo1
 * Ejecutar con:    mpirun -np X Ejemplo1
 * Donde X es un número entero mayor que 0.
*****************************************************************************/

#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv)
{
    int myrank, nprocs;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    printf("Hola soy el proceso %d de un total de %d\n", myrank, nprocs);
    MPI_Finalize();
    return 0;
}
```

En este programa podemos ver rutinas simples y comunes a gran multitud de programas como son, aparte de la necesaria ***MPI_Init***, ***MPI_Comm_size*** y ***MPI_Comm_rank***. La primera nos permite conocer el tamaño del universo de comunicación del comunicador que se le pasa como argumento, es decir, el número de procesos que se encuentran dentro de él. La segunda nos permite conocer el rango del proceso que la llama dentro del comunicador pasado como argumento.

Tras esto, cada proceso muestra su rango dentro del comunicador **MPI_COMM_WORLD**, finalizando el programa con ***MPI_Finalize***.

El comunicador usado en las anteriores funciones es el comunicador que se crea por defecto, **MPI_COMM_WORLD**, y que contiene a todos los procesos creados simultáneamente de forma estática. Como veremos más adelante,

cada vez que se cree un nuevo conjunto de procesos, estos se podrán comunicar mediante su propio **MPI_COMM_WORLD**, que a pesar de denominarse igual, será un comunicador que delimita un universo de comunicación completamente distinto (el ejemplo más claro es con el uso de ***MPI_Comm_spawn***).

De esta manera al crear estos X procesos estáticamente, están todos englobados dentro del **MPI_COMM_WORLD** y se pueden comunicar entre ellos, punto a punto o de forma colectiva, a través del mismo.

Hay que tener en cuenta que las operaciones colectivas engloban a todos los procesos dentro del intracomunicador (o intercomunicador). Si alguno de ellos no llamara a la función colectiva, el programa quedaría bloqueado. En el siguiente ejemplo se muestra un ejemplo donde se hacen uso de las funciones colectivas. Un proceso crea una serie de valores, los reparte con ***MPI_Scatter*** entre todos los procesos, y luego, tras multiplicar cada proceso el valor recibido por -1, los vuelve a recoger el mismo proceso con ***MPI_Gather***.

```
/
******************************************************************************
 * Fichero: Ejemplo2.c
 * Ejemplo de funcionamiento de rutinas colectivas con Gather y Scatter
 * El proceso 0 manda un valor aleatorio al resto de procesos
 * estos lo multiplican por -1 y lo devuelven al maestro.
 *
 * Compilar con:    mpicc Ejemplo2.c -o Ejemplo2
 * Ejecutar con:    mpirun -np X Ejemplo2
 * X puede ser cualquier número entero positivo mayor que 1

******************************************************************************/

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

int main(int argc, char** argv)
{
    //Declaración de variables a usar en el programa
    int myrank, nprocs, i;
    int *valores;
    int recibido;
    int *valoresfinal;

    //Inicialización de MPI, cada proceso obtiene en nº de procesos y su rango
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    // Semilla para generar secuencias aleatorias
    time_t t;
    srand((unsigned) time(&t));

    valoresfinal = (int *)malloc(nprocs*sizeof(int));
```

```
    //Si solo existe un proceso, el programa finaliza
    if (nprocs==1) {
      printf(" Solo existe un proceso por lo que finaliza el ejemplo. \n");
      MPI_Finalize();
      return 0;
    }

    //Reservamos memoria y generamos el vector de valores a repartir
    valores = (int *) malloc(nprocs*sizeof(int));
    if (myrank == 0){
      for (i=0; i<nprocs; i++) {
          valores[i] = 1+rand()%100;
      }
    }

    //Mandamos un valor de los generados a cada proceso
    MPI_Scatter(valores, 1, MPI_INT, &recibido, 1, MPI_INT, 0,
MPI_COMM_WORLD);
    printf("Hola desde el proceso de rango %d de un total de %d\n", myrank,
nprocs);
    printf("Soy el proceso %d y he recibido %d del maestro \n", myrank,
recibido);
    recibido = recibido*(-1);
    //Se reciben los valores modificados por parte del proceso 0,
    //que es el root
    MPI_Gather(&recibido, 1, MPI_INT, valoresfinal, 1, MPI_INT, 0,
MPI_COMM_WORLD);

    if (myrank == 0){
      for (i=1; i<nprocs; i++) {
          printf("Hola, soy %d y he recibido del proceso %d el valor %d \n",
myrank, i, valoresfinal[i]);
      }
    }

    MPI_Finalize();
    return 0;
}
```

Para que el programa funcione es necesario que todos los procesos pertenecientes al comunicador involucrado, **MPI_COMM_WORLD** en este caso, llamen a las funciones colectivas en el orden adecuado, aparte, por supuesto, de que el resto de parámetros de las funciones concuerden de manera correcta para que la comunicación sea posible (buffers de envío y recepción correcto, número de datos enviados y recibidos correctos, tipo de datos...).

Si quisiéramos que se comunicaran los datos, por ejemplo, sólo a los procesos con rangos impares, o bien usábamos comunicaciones punto a punto, que sería una solución poco eficiente en el caso de comunicaciones múltiples, o bien usaríamos funciones colectivas como en el ejemplo anterior. Sin embargo, si usáramos el **MPI_COMM_WORLD** en las colectivas, el programa sería erróneo, ya que como hemos dicho, todos los procesos del comunicador deben llamar a las funciones colectivas, ya que si

sólo un subconjunto del comunicador las llama, el programa quedará bloqueado.

Para poder comunicar a un subgrupo de procesos con funciones colectivas debemos crear un nuevo comunicador que sólo involucre a los procesos que nos interesan. Para esto vamos a ver el comportamiento de las diferentes funciones para crear grupos y comunicadores nuevos basándonos en el ejemplo anterior.

Como primer ejemplo de esto vamos a crear un grupo con los procesos pares y otro con los procesos impares, pero sin comunicador asociado. Veamos un ejemplo de lo comentado anteriormente.

```
/
*****************************************************************************
 * Fichero: GruposIntracomunicadores.c
 *
 * Compilar con:    mpicc GruposIntracomunicadores.c -o intra1
 * Ejecutar con:    mpirun -np X intra1
 * siend X un número entero mayor que 1


 * Creamos de forma estática un conjunto de procesos, se crearán dos grupos
 * de procesos, uno para los de rango par y otro para los de impar. La
 * creación de grupos no es colectiva, por lo que se muestran dos alternativas
 * para crear los grupos, una comentada y otra sin comentar.
*****************************************************************************/

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

int main(int argc, char** argv)
{
    int myrank, nprocs;
    int gproc, n_par, n_impar;
    int i, *rangos_par, *rangos_impar;
    int rankpar, rankimpar;
    MPI_Group grupo_completo, grupo_par, grupo_impar;

    // Inicializamos MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    // Creamos un grupo con todos los procesos para luego dividirlo
    MPI_Comm_group(MPI_COMM_WORLD, &grupo_completo);
    MPI_Group_size(grupo_completo, &gproc);
    printf(" El grupo completo %d elementos \n", gproc);

    //Determinamos nº elementos y rangos de cada grupo
    rangos_par = (int *) malloc(nprocs*sizeof(int));
    rangos_impar = (int *) malloc(nprocs*sizeof(int));
    n_par = 0;
    n_impar = 0;
    for (i=0; i<nprocs; i++) {
```

```
      if (i%2==0){
          rangos_par[n_par] = i;
          n_par = n_par+1;
      } else {
          rangos_impar[n_impar] = i;
          n_impar = n_impar+1;
      }
    }

    // Creamos los grupos llamando todos los procesos a todas las funciones
    // COMENTADO

    /* MPI_Group_incl(grupo_completo, n_par, rangos_par, &grupo_par);
      MPI_Group_incl(grupo_completo, n_impar, rangos_impar, &grupo_impar);

      MPI_Group_size(grupo_par, &n_par);
      MPI_Group_rank(grupo_par, &rankpar);
      MPI_Group_size(grupo_impar, &n_impar);
      MPI_Group_rank(grupo_impar, &rankimpar);

      printf(" Soy %d. El grupo par %d elementos y tengo rango %d \n",
myrank, n_par, rankpar);
      printf(" Soy %d. El grupo impar %d elementos y tengo rango %d \n",
myrank, n_impar, rankimpar);
    */

    // Creamos los grupos llamando sólo cada proceso a las funciones para
    // crear su grupo
    if (myrank%2==0){
      MPI_Group_incl(grupo_completo, n_par, rangos_par, &grupo_par);
      MPI_Group_size(grupo_par, &n_par);
      MPI_Group_rank(grupo_par, &rankpar);
      printf(" Soy %d. El grupo par %d elementos y tengo rango %d \n",
myrank, n_par, rankpar);

    } else {
      MPI_Group_incl(grupo_completo, n_impar, rangos_impar, &grupo_impar);
      MPI_Group_size(grupo_impar, &n_impar);
      MPI_Group_rank(grupo_impar, &rankimpar);
      printf(" Soy %d. El grupo impar %d elementos y tengo rango %d \n",
myrank, n_impar, rankimpar);
    }

    MPI_Finalize();
    return 0;
}
```

En el código anterior se muestran dos alternativas para crear los grupos (una de ellas comentada para que el código funcione adecuadamente). En la primera todos los procesos llaman dos veces a ***MPI_Group_incl***, una para el grupo par y otra para el impar. De esta manera, si todos los procesos ejecutan la función ***MPI_Group_rank*** para saber su rango dentro de cada grupo, dará el rango correcto dentro del grupo al que pertenece el proceso en cuestión y un valor espúreo para el otro, ya que los miembros de cada grupo se han especificado con los rangos del original pasados como parámetro de la función de creación de grupos.

Sin embargo, estas funciones son locales, y no es necesario que todos los procesos del comunicador original o del grupo original las llamen, sólo aquellos que van a pertenecer al nuevo grupo. De esta manera, en la segunda opción, cada proceso ejecuta únicamente la función para crear su grupo, siendo el funcionamiento igualmente correcto.

Esto no es así en el caso de querer crear nuevos comunicadores basándonos en los grupos o en los comunicadores existentes, ya que todos los procesos del comunicador origen deben hacer la llamada para crear el nuevo comunicador. Esto lo vamos a ver en los siguientes ejemplos donde creamos comunicadores nuevos para cada uno de los grupos creados.

```
/*****************************************************************************
 * Fichero: GruposIntracomunicadores2.c
 *
 * Compilar con:    mpicc GruposIntracomunicadores2.c -o intra2
 * Ejecutar con:    mpirun -np X intra2
 * siendo X un número entero mayor que 1. Preferiblemente impar para ver
 * la diferencia en el número de procesos por intracomunicador

 * Creamos de forma estática un conjunto de procesos, se crearán dos grupos
 * de procesos, uno para los de rango par y otro para los de impar. Una vez
 * creados los grupos, se creará un intracomunicador para cada uno. En este
 * caso la creación de los comunicadores sí es colectiva, por lo que todos los
 * procesos del comunicador original deben llamar a la función de creación
 * de los nuevos, vayan a pertenecer o no a ellos. Sin embargo una vez creados
 * sólo los procesos que pertenecen a los comunicadores pueden hacer llamadas
 * a ellos
*****************************************************************************/

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

int main(int argc, char** argv)
{
    int myrank, nprocs;
    int gproc, n_par, n_impar;
    int i, *rangos_par, *rangos_impar;
    int rankpar, rankimpar;
    MPI_Comm comPar, comImpar;
    MPI_Group grupo_completo, grupo_par, grupo_impar;

    // Inicializamos MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    // Creamos un grupo con todos los procesos para luego dividirlo
    MPI_Comm_group(MPI_COMM_WORLD, &grupo_completo);
    MPI_Group_size(grupo_completo, &gproc);
    printf(" El grupo completo %d elementos \n", gproc);

    //Determinamos nº elementos y rangos de cada grupo
    rangos_par = (int *) malloc(nprocs*sizeof(int));
    rangos_impar = (int *) malloc(nprocs*sizeof(int));
    n_par = 0;
```

```
    n_impar = 0;
    for (i=0; i<nprocs; i++) {
       if (i%2==0){
           rangos_par[n_par] = i;
           n_par = n_par+1;
       } else {
           rangos_impar[n_impar] = i;
           n_impar = n_impar+1;
       }
    }

    // Creamos los grupos par e impar
    MPI_Group_incl(grupo_completo, n_par, rangos_par, &grupo_par);
    MPI_Group_incl(grupo_completo, n_impar, rangos_impar, &grupo_impar);
    // Creamos los nuevos intracomunicadores de procesos pares e impares
    // a partir del MPI_COMM_WORLD, por lo que todos los miembros de
    // MPI_COMM_WORLD deben llamar a las funciones.
    MPI_Comm_create(MPI_COMM_WORLD, grupo_par, &comPar);
    MPI_Comm_create(MPI_COMM_WORLD, grupo_impar, &comImpar);

    if (myrank%2==0){ // procesos pares
       // Comprobamos tamaño y rango del grupo par.
       MPI_Group_size(grupo_par, &n_par);
       MPI_Group_rank(grupo_par, &rankpar);
       printf(" Soy %d. El grupo par %d elementos y tengo rango %d \n",
myrank, n_par, rankpar);
       // Comprobamos tamaño y rango del intracomunicador par. Estas funciones
       // sólo pueden llamarlas los procesos que pertenecen a dichos
       // comunicadores
       MPI_Comm_size(comPar, &n_par);
       MPI_Comm_rank(comPar, &rankpar);
       printf(" Soy %d. El comunicador par %d elementos y tengo rango %d \n",
myrank, n_par, rankpar);
    } else {
       // Comprobamos tamaño y rango del grupo impar.
       MPI_Group_size(grupo_impar, &n_impar);
       MPI_Group_rank(grupo_impar, &rankimpar);
       printf(" Soy %d. El grupo impar %d elementos y tengo rango %d \n",
myrank, n_impar, rankimpar);
       // Comprobamos tamaño y rango del intracomunicador par. Estas funciones
       // sólo pueden llamarlas los procesos que pertenecen a dichos
       // comunicadores
       MPI_Comm_size(comImpar, &n_impar);
       MPI_Comm_rank(comImpar, &rankimpar);
       printf(" Soy %d. El comunicador impar %d elementos y tengo rango %d
\n", myrank, n_impar, rankimpar);
    }

    MPI_Finalize();
    return 0;
}
```

En este ejemplo todos los procesos deben llamar a la función de creación de los comunicadores, sea para el nuevo comunicador del que formarán parte como para el que no. Esto es por lo que hemos comentado anteriormente, todos los procesos del comunicador original deben llamar a la función que crea el nuevo comunicador vayan a pertenecer o no a dicho nuevo comunicador.

Las funciones ***MPI_Comm_rank*** y ***MPI_Comm_size*** solo pueden llamarse en cada proceso para los comunicadores a los que pertenece, ya que aunque todos los procesos hayan llamado a las funciones de creación de comunicadores, estos sólo son visibles y pueden ser llamados por los procesos que finalmente forman parte de estos nuevos comunicadores. Así, aquellos procesos que no formen parte del comunicador **Pares** pero lo llamen devolverán error, y lo mismo para los que no pertenecen a **Impares**.

El anterior ejemplo es una manera de crear dos nuevos comunicadores. Los grupos se han hecho disjuntos y con el total de procesos, pero no tiene por qué ser así, puede haber procesos que estén en todos los nuevos comunicadores, en uno sólo o en ninguno. Como hemos comentado, cada comunicador es un universo de comunicación independiente y puede estar compuesto de cualquier número de procesos, por lo que una comunicación usando un comunicador no afecta a otras comunicaciones con otros comunicadores. Veamos cómo sería y cómo no afectaría al correcto funcionamiento del programa. Para ello creamos un grupo de procesos con los rangos pares, y un grupo con aquellos cuyos rangos son múltiplos de 3.

```
/
*****************************************************************************
 * Fichero: GruposIntracomunicadores3.c
 *
 * Compilar con:    mpicc GruposIntracomunicadores3.c -o intra3
 * Ejecutar con:    mpirun -np X intra3
 * siendo X un número entero mayor que 1. Preferiblemente impar para ver
 * la diferencia en el número de procesos por intracomunicador

 * Creamos de forma estática un conjunto de procesos, se crearán dos grupos
 * de procesos, uno para los de rango par y otro para los múltiplos de tres
 * para ver cómo crear comunicadores que se solapan en los procesos. Una vez
 * creados los grupos, se creará un intracomunicador para cada uno. En este
 * caso la creación de los comunicadores sí es colectiva, por lo que todos los
 * procesos del comunicador original deben llamar a la función de creación
 * de los nuevos, vayan a pertenecer o no a ellos.

*****************************************************************************/

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

int main(int argc, char** argv)
{
    int myrank, nprocs;
    int gproc, n_par, n_tres;
    int i, *rangos_par, *rangos_tres;
    int rankpar, ranktres;
    MPI_Comm comPar, comTres;
    MPI_Group grupo_completo, grupo_par, grupo_tres;

    // Inicializamos MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

```
    // Creamos un grupo con todos los procesos para luego dividirlo
    MPI_Comm_group(MPI_COMM_WORLD, &grupo_completo);
    MPI_Group_size(grupo_completo, &gproc);
    printf(" El grupo completo %d elementos \n", gproc);

    //Determinamos nº elementos y rangos de cada grupo
    rangos_par = (int *) malloc(nprocs*sizeof(int));
    rangos_tres = (int *) malloc(nprocs*sizeof(int));
    n_par = 0;
    n_tres = 0;
    for (i=0; i<nprocs; i++) {
       if (i%2==0){
           rangos_par[n_par] = i;
           n_par = n_par+1;
       }
       if (i%3==0) {
           rangos_tres[n_tres] = i;
           n_tres = n_tres+1;
       }
    }

    // Creamos los grupos e intracomunicadores
    MPI_Group_incl(grupo_completo, n_par, rangos_par, &grupo_par);
    MPI_Comm_create(MPI_COMM_WORLD, grupo_par, &comPar);
    MPI_Group_incl(grupo_completo, n_tres, rangos_tres, &grupo_tres);
    MPI_Comm_create(MPI_COMM_WORLD, grupo_tres, &comTres);

    if (myrank%2==0){
       MPI_Group_size(grupo_par, &n_par);
       MPI_Group_rank(grupo_par, &rankpar);
       printf(" Soy %d. El grupo par %d elementos y tengo rango %d \n",
myrank, n_par, rankpar);
       MPI_Comm_size(comPar, &n_par);
       MPI_Comm_rank(comPar, &rankpar);
       printf(" Soy %d. El comunicador par %d elementos y tengo rango %d \n",
myrank, n_par, rankpar);
    }

    // En este caso debe haber dos if ya que hay procesos que pueden estar
    // en ambos comunicadores
    if (myrank%3==0){
       MPI_Group_size(grupo_tres, &n_tres);
       MPI_Group_rank(grupo_tres, &ranktres);
       printf(" Soy %d. El grupo TRES %d elementos y tengo rango %d \n",
myrank, n_tres, ranktres);
       MPI_Comm_size(comTres, &n_tres);
       MPI_Comm_rank(comTres, &ranktres);
       printf(" Soy %d. El comunicador TRES %d elementos y tengo rango %d \n",
myrank, n_tres, ranktres);
    }

    MPI_Finalize();
    return 0;
}
```

En este caso hay procesos que están en los dos intracomunicadores, otros sólo en uno, y otros en ninguno, por lo que podemos apreciar la gran posibilidad de entornos o topologías de comunicación que podemos crear.

En los ejemplos anteriores hemos visto como crear los intracomunicadores nuevos a partir de grupos que se han creado previamente. Otra alternativa, en la que no se usarían las funciones de manejo de grupos, es usar la función ***MPI_Comm_split*** que nos permite crear diversos comunicadores de manera simultánea si son disjuntos, o de forma recursiva si hay procesos que están en varios comunicadores. Veamos en primer lugar cuando queremos crear dos intracomunicadores disjuntos, uno para los procesos pares y otro para los procesos impares. En estos ejemplos las funciones ***MPI_Comm_set_name*** y ***MPI_Comm_get_name*** simplemente se utilizan para definir un array de caracteres como nombre identificativo de los nuevos comunicadores y poder distinguir mejor qué comunicadores tiene cada proceso.

```
/
*****************************************************************************
 * Fichero: IntraSplit.c
 *
 * Compilar con:    mpicc IntraSplit.c -o split1
 * Ejecutar con:    mpirun -np X split1
 * siendo X un número entero mayor que 1. Preferiblemente impar para ver
 * la diferencia en el número de procesos por intracomunicador

 * Creamos de forma estática un conjunto de procesos, y se crearán
 * dos intracomunicadores directamente con MPI_Comm_split. Todos los
 * procesos deben llamar a Split ya que es colectiva, y para simplificar
 * el código se ha utilizado la misma variable para los dos
 * intracomunicadores, aunque son dos diferentes comunicadores,
 * uno para pares y otro para impares.
*****************************************************************************/

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

int main(int argc, char** argv){

    int myrank, nprocs;
    int gproc;
    int rankparimpar;
    int color;
    MPI_Comm comParImpar;
    char *name;
    int len = 0;

    // Inicializamos MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    name = (char *)malloc(57*sizeof(char));
```

```
    // con color determinamos a qué comunicador pertenecerá cada proceso
    color = myrank%2;
    // Dividimos el MPI_COMM_WORLD en dos intracomunicadores,
    // ambos con la misma variable
    MPI_Comm_split(MPI_COMM_WORLD, color, myrank, &comParImpar);
    if (myrank%2 == 0)
      MPI_Comm_set_name(comParImpar,"comPar");
    else
      MPI_Comm_set_name(comParImpar,"comImpar");
    // Cada proceso obtiene el tamaño de su intracomunicador y que rango
    // tiene dentro de él
    MPI_Comm_size(comParImpar, &gproc);
    MPI_Comm_rank(comParImpar, &rankparimpar);
    MPI_Comm_get_name(comParImpar,name,&len);
    printf("Soy %d, y dentro de mi nuevo comunicador (%s) de tamaño %d soy
%d \n",myrank, name, gproc, rankparimpar);

    MPI_Finalize();
    return 0;
}
```

En este caso todos los procesos llaman a la misma función ***MPI_Comm_split***, tan sólo el argumento *color* cambia dependiendo del proceso que la llame, y según su valor indicará a qué comunicador pertenece. Es importante darse cuenta que en este caso se ha usado la misma variable **comParImpar** para el comunicador tanto de los procesos pares como de los impares, sin embargo son dos comunicadores separados y diferentes, uno para los pares y otro para los impares, y solo podrán usarse para comunicaciones entre procesos pares o entre procesos impares, no entre un par y un impar. Para esta última comunicación se usaría **MPI_COMM_WORLD** que es el comunicador que reúne tanto a pares como impares. A modo de ejemplo para verlo más claro, podemos pensar en el uso de la función ***MPI_Comm_spawn***, los procesos padres tienen un **MPI_COMM_WORLD**, y los hijos tienen otro propio, pero son dos comunicadores diferentes.

Una alternativa, si queremos que tengan variables diferentes, es separar la llamada según procesos y cada uno le dé una variable al comunicador. Esto lo podemos ver en el siguiente ejemplo:

```
/
*************************************************************************
 * Fichero: IntraSplit2.c
 *
 * Compilar con:    mpicc IntraSplit2.c -o split2
 * Ejecutar con:    mpirun -np X split2
 * siendo X un número entero mayor que 1. Preferiblemente impar para ver
 * la diferencia en el número de procesos por intracomunicador

 * Creamos de forma estática un conjunto de procesos, y se crearán
 * dos intracomunicadores directamente con MPI_Comm_split. Todos los
 * procesos deben llamar a Split ya que es colectiva.
 * En este caso se le asigna a cada intracomunicador una variable
 * diferente, para mostrar que es exactamente igual usar la misma o no,
```

```
 * y se crean dos intracomunicadores en función del parámetro color.

*************************************************************************/

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

int main(int argc, char** argv){

    int myrank, nprocs;
    int gproc;
    int rankparimpar;
    int color;
    MPI_Comm comPar, comImpar;
    char *namep;
    char *namei;
    int len = 0;

    // Inicializamos MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    namep = (char *)malloc(57*sizeof(char));
    namei = (char *)malloc(57*sizeof(char));

    // Con color determinamos a qué intracomunicador pertenece cada proceso
    color = myrank%2;
    // Los procesos pares le darán una variable a su comunicador y los impares
    // otra pero todos los procesos del comunicador origen, MPI_COMM_WORLD
    // debenllamar a split aunque cada uno le dé una variable a su
    // comunicador.
    if (color==0){
       // Los pares crean su comunicador
       MPI_Comm_split(MPI_COMM_WORLD, color, myrank, &comPar);
       MPI_Comm_set_name(comPar,"comPar");
       // Cada proceso obtiene el tamaño de su intracomunicador y que rango
       // tiene dentro de él
       MPI_Comm_size(comPar, &gproc);
       MPI_Comm_rank(comPar, &rankparimpar);
       MPI_Comm_get_name(comPar,namep,&len);
       printf("Soy %d, par, y dentro de mi nuevo comunicador (%s) de tamaño %d
soy %d \n",myrank, namep, gproc, rankparimpar);
    }else{
       // Los impares crean su comunicador
       MPI_Comm_split(MPI_COMM_WORLD, color, myrank, &comImpar);
       MPI_Comm_set_name(comImpar,"comImpar");
       // Cada proceso obtiene el tamaño de su intracomunicador y que rango
       // tiene dentro de él
       MPI_Comm_size(comImpar, &gproc);
       MPI_Comm_rank(comImpar, &rankparimpar);
       MPI_Comm_get_name(comImpar,namei,&len);
       printf("Soy %d, impar, y dentro de mi nuevo comunicador (%s) de tamaño
%d soy %d \n",myrank, namei, gproc, rankparimpar);
    }

    MPI_Finalize();
    return 0;
}
```

Este ejemplo es exactamente igual que el anterior, pero ahora los pares le dan una variable a su comunicador (**comPar**) y los impares otra (**comImpar**), pero el funcionamiento y resultado es exactamente el mismo, pero el código es mayor.

Al igual que con ***MPI_Comm_create*** podemos hacer que los comunicadores no sean disjuntos haciendo múltiples llamadas consecutivas a ***MPI_Comm_split***, o si queremos que sólo unos pocos procesos estén en el comunicador, como se puede ver en el siguiente ejemplo.

```
/
*****************************************************************************
 * Fichero: IntraSplit3.c
 *
 * Compilar con:    mpicc IntraSplit3.c -o split3
 * Ejecutar con:    mpirun -np X split3
 * siendo X un número entero mayor que 1. Preferiblemente impar para ver
 * la diferencia en el número de procesos por intracomunicador

 * Creamos de forma estática un conjunto de procesos, y se crearán
 * dos intracomunicadores directamente con MPI_Comm_split, pero en este caso
 * los procesos podrán pertenecer a los dos, a uno o a ningún comunicador.
 * Se creará uno para procesos pares y otro para múltiplos de 3. Todos
 * los procesos deben llamar a Split ya que es colectiva. En este caso se
 * deben hacer dos llamadas a split por cada proceso, ya que los comunicadores
 * no son disjuntos. Aquel proceso que no esté en un determinado comunicador
 * pasará en el parámetro color el valor MPI_UNDEFINED en split.

*****************************************************************************/

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

int main(int argc, char** argv){

    int myrank, nprocs;
    int gproc;
    int rankcomm;
    int color, color3;
    MPI_Comm comPar, comMult3;
    char *namep;
    char *namem;
    int len = 0;

    // Inicializamos MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    namem = (char *)malloc(57*sizeof(char));
    namep = (char *)malloc(57*sizeof(char));

    // Creamos el primer comunicador para los procesos pares, y asignamos
    // a color el mismo valor para todos los procesos pares.
```

```
    color = myrank%2;
    if (color==0){
       // Si el proceso es par, crea su comunicador de procesos pares
       MPI_Comm_split(MPI_COMM_WORLD, color, myrank, &comPar);
       MPI_Comm_set_name(comPar,"compar");
       // Cada proceso par obtiene el tamaño de su intracomunicador y qué
       // rango tiene dentro de él
       MPI_Comm_size(comPar, &gproc);
       MPI_Comm_rank(comPar, &rankcomm);
       MPI_Comm_get_name(comPar,namep,&len);
       printf("Soy %d, par, y dentro de mi nuevo comunicador (%s) de tamaño %d
soy %d \n",myrank, namep, gproc, rankcomm);
    }else{
       // Los impares también deben llamar a split pero no crean comunicador
       // para ello en color se pasa MPI_UNDEFINED
       MPI_Comm_split(MPI_COMM_WORLD, MPI_UNDEFINED, myrank, &comPar);
    }
    // Ahora se crea un intracomunicador para los múltiplos de 3
    color3 = myrank%3;
    if (color3==0){
       // Los múltiplos de 3 crean su comunicador
       MPI_Comm_split(MPI_COMM_WORLD, color3, myrank, &comMult3);
       MPI_Comm_set_name(comMult3,"mult3");
       // Cada proceso para obtiene el tamaño de su intracomunicador y qué
       // rango tiene dentro de él
       MPI_Comm_size(comMult3, &gproc);
       MPI_Comm_rank(comMult3, &rankcomm);
       MPI_Comm_get_name(comMult3,namem,&len);
       printf("Soy %d, multiplo de 3, y dentro de mi nuevo comunicador (%s) de
tamaño %d soy %d \n",myrank, namem, gproc, rankcomm);
    } else{
       // El resto de procesos llama a split con MPI_UNDEFINED en color
       // ya que no van a crear comunicador
       MPI_Comm_split(MPI_COMM_WORLD, MPI_UNDEFINED, myrank, &comMult3);
    }

    MPI_Finalize();
    return 0;
}
```

En este caso queremos crear un comunicador para los procesos que sean múltiplos de 2, y otro para los que sean múltiplos de 3. De esta manera habrá procesos que estén en los dos intracomunicadores, otros que sólo estén en uno, y otros que no estén en ninguno. Los que queremos que estén en el intracomunicador de los pares pasarán el mismo valor en el parámetro *color*. Los demás que no sean pares pasarán el valor *MPI_UNDEFINED*, indicando que no van a crear comunicador aparte. Como se puede ver la llamada es colectiva, y todos los procesos del comunicador original deben llamarla. Para el segundo intracomunicador se procede de la misma manera, los múltiplos de 3 pasarán el mismo valor de *color*, y los demás *MPI_UNDEFINED*, y tendremos así un intracomunicador con los procesos pares y otro con los procesos múltiplos de 3.

2. INTERCOMUNICADORES

Hasta el momento hemos visto ejemplos de cómo crear intracomunicadores a partir de un intracomunicador inicial, que en nuestro caso ha sido **MPI_COMM_WORLD**. Ahora vamos a pasar a ver cómo podemos trabajar con intercomunicadores, y a partir de ellos crear intra e intercomunicadores.

En primer lugar, debemos tener claros que los intercomunicadores comunican dos conjuntos de procesos separados, que llamaremos grupos A y B. Al igual que con los intracomunicadores, podremos hacer tantos intercomunicadores como queramos y con los miembros que queramos en cada grupo a partir de los comunicadores existentes. Tenemos que tener en cuenta que como mínimo siempre tendremos el **MPI_COMM_WORLD**. De esta manera, los intercomunicadores nos darán una vía de comunicación para comunicar los procesos de A con los de B y viceversa. Para comunicarse entre sí los de A o los de B usarán un intracomunicador (o intercomunicador en donde los procesos que se quieran comunicar estén en grupos diferentes).

La manera más evidente de tener un intercomunicador es con la rutina ***MPI_Comm_spawn***, donde se creará un grupo de procesos hijos junto con un intercomunicador que comunicará los procesos padres e hijos. Hay que tener en cuenta que ***MPI_Comm_spawn*** es una llamada colectiva, de tal manera que si se llama sobre **MPI_COMM_WORLD**, todos los procesos del comunicador deben llamar a ***MPI_Comm_spawn*** aunque sólo uno de ellos sea el raíz. Para que únicamente exista un proceso padre, ***MPI_Comm_spawn*** debe llamarse sobre el comunicador **MPI_COMM_SELF**, que como se indica en el libro base de la asignatura es un comunicador que sólo contiene al proceso que lo llama, por lo que sólo el proceso padre y raíz llamará a ***MPI_Comm_spawn*** en este caso.

Para ver el funcionamiento de un intercomunicador, vamos a ver el siguiente ejemplo, en el que un proceso padre es generado de forma estática, y este proceso padre crea a los procesos hijos. El proceso padre mandará a cada uno de los hijos un valor aleatorio con comunicaciones punto a punto, y los hijos multiplicarán este valor por otro valor aleatorio. Los resultados de los hijos los recogerá el padre con la función colectiva ***MPI_Gather***, y luego buscará el mayor valor obtenido de los procesos hijos pares y el menor de los impares.

```
/
******************************************************************************
 * Fichero: Intercomunicadores1.c
 *
 * Compilar con:    mpicc Intercomunicadores1.c -o inter1
 * Ejecutar con:    mpirun -np 1 inter1

 * Enunciado:

 * Un proceso maestro genera de manera dinámica un número de procesos.
 * A su vez, este proceso maestro generará números aleatorios y se los
```

```
 * enviará a los procesos con rango par,  que lo multiplicarán por otro
 * número aleatorio. Luego el maestro creará otros números aleatorios y
 * se los enviará a los procesos de rango impar, que también lo multiplicarán
 * por un número aleatorio. Finalmente el proceso maestro deberá conocer
 * cuál es el mayor valor resultante de las multiplicaciones en
 * cada uno de los procesos de rango par, y el menor valor de entre los
 * procesos de rango impar.
 * NOTA: EL número de procesos hijos se considera par por simplicidad
*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "mpi.h"

#define PROCESOS 10

/*****************************************************************************
 * Programa principal
*****************************************************************************/
int main(int argc, char *argv[]) {
    // Declaración de variables
    int procesos;
    int miRango;
    MPI_Comm intercom;
    time_t t;
    float numeros[PROCESOS];
    int i, maximo, minimo;
    float Valor;

    // Inicia MPI
    MPI_Init(&argc, &argv);

    // Obtiene su rango y número de procesos
    MPI_Comm_size(MPI_COMM_WORLD, &procesos);
    MPI_Comm_rank(MPI_COMM_WORLD, &miRango);


    // Semilla para generar secuencias aleatorias
    srand((unsigned) time(&t));


    MPI_Comm_get_parent(&intercom); // Para conocer si tiene padre o no

    if (intercom == MPI_COMM_NULL) {
       // Se crean los procesos hijos
       if (procesos>1) {
           printf(" Hay más de un proceso maestro, por lo que es erróneo. Se
cierra el programa. \n");
           MPI_Finalize();
           return -1;
       }
       MPI_Comm_spawn(argv[0], MPI_ARGV_NULL, PROCESOS, MPI_INFO_NULL, 0,
MPI_COMM_SELF, &intercom, MPI_ERRCODES_IGNORE);

       // Se envían los datos, primero a pares, luego a los impares.
      // Comunicación punto a punto.
       for (i=0; i<PROCESOS/2; i++) {
           Valor = 1+rand()%100;
           MPI_Send(&Valor, 1, MPI_FLOAT, i*2, 2, intercom);
       }
```

```
       for (i=0; i<PROCESOS/2; i++) {
           Valor = 1+rand()%100;
           MPI_Send(&Valor, 1, MPI_FLOAT, i*2+1, 1, intercom);
       }
       // Hacemos un gather para recoger los valores y
      // luego buscamos entre pares e impares
       MPI_Gather(&Valor, 1, MPI_FLOAT, numeros, 1, MPI_FLOAT, MPI_ROOT,
intercom);

       // Ahora recorremos el vector de números y buscamos los valores.
       maximo = 0;
       minimo = 10000;
       for (i=0; i<PROCESOS/2; i++) {
           if (numeros[2*i]>maximo) maximo = numeros[2*i];
       }
       for (i=0; i<PROCESOS/2; i++) {
           if (numeros[2*i+1]<minimo) minimo = numeros[2*i+1];
       }
       printf("El valor máximo de los procesos pares es %d.\n", maximo);
       printf("El valor mínimo de los procesos impares es %d.\n", minimo);

    } else { //no es el proceso padre, es un hijo
       // Se reciben los valores y se multiplica por un valor aleatorio
       MPI_Recv(&Valor, 1, MPI_FLOAT, 0, MPI_ANY_TAG, intercom,
MPI_STATUS_IGNORE);
       Valor = Valor*(rand()%100);
       printf("Valor = %f del proceso %d \n",Valor, miRango);
       // Hacemos un gather para enviar los valores al padre
      // a través del intercomunicador
       MPI_Gather(&Valor, 1, MPI_FLOAT, numeros, 1, MPI_FLOAT, 0, intercom);

    } //fin del if para distinguir el proceso original

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

Vamos a ver un segundo ejemplo en donde la comunicación sea un poco diferente. En el siguiente ejemplo, el proceso maestro (creado de manera estática el solo) genera un número de procesos y les pasa con comunicación punto a punto un valor aleatorio. Los hijos se dividen en dos intracomunicadores, uno para los procesos con rango par y otro para los procesos de rango impar. Ambos comunicadores se llaman igual, pero designan grupos diferentes por lo que no hay ningún problema de comunicación entre ellos. Luego, con la función ***MPI_Allgather***, cada proceso recoge los valores de todos los procesos de su nuevo intracomunicador, usando los intracomunicadores creados para cada grupo, y posteriormente los suma (sería algo equivalente a haberlo hecho con un ***MPI_Allreduce***, podéis probarlo modificando el código del ejemplo). Tras esto todos los procesos mandan el valor obtenido con un ***MPI_Gather*** al maestro a través del intercomunicador creado con ***MPI_Comm_spawn***. El padre finalmente realiza la suma de los valores de los hijos pares, y la resta de los de los impares, mostrando los resultados por salida estándar.

```
/
*****************************************************************************
 * Fichero: Intercomunicadores2.c
 *
 * Compilar con:    mpicc Intercomunicadores2.c -o inter2
 * Ejecutar con:    mpirun -np 1 inter2

 * Enunciado:

 * Un proceso maestro genera de manera dinámica un número de procesos.
 * A su vez, este proceso maestro generará números aleatorios y se los
 * enviará a los procesos con rango par,  que lo multiplicarán por
 * otro número aleatorio. Luego el maestro creará otros números aleatorios y
 * se los enviará a los procesos de rango impar, que también lo
 * multiplicarán por un número aleatorio.  Cada proceso recogerá los
 * valores de su grupo (par o impar), los sumará y enviará al padre. El
 * padre sumará los valores enviados por los pares y restará los enviados
 * por los impares.

 * NOTA: EL número de procesos hijos se considera par por simplicidad
  ****************************************************************************
/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "mpi.h"

#define PROCESOS 10

/
*****************************************************************************
 * Programa principal
  ****************************************************************************
/
int main(int argc, char *argv[]) {
    // Declaración de variables
    int procesos;
    int miRango;
    MPI_Comm intercom, ParesImpares;
    time_t t;
    int numeros[PROCESOS];
    int i, Suma, Resta;
    int Valor, *valores;
    int color;
    int procesosdivididos;

    // Inicia MPI
    MPI_Init(&argc, &argv);

    // Obtiene su rango y número de procesos
    MPI_Comm_size(MPI_COMM_WORLD, &procesos);
    MPI_Comm_rank(MPI_COMM_WORLD, &miRango);


    // Semilla para generar secuencias aleatorias
    srand((unsigned) time(&t));


    MPI_Comm_get_parent(&intercom); // Para conocer si tiene padre o no
```

```
    if (intercom == MPI_COMM_NULL) {
      // Se crean los procesos hijos

      if (procesos>1) {
           printf(" Hay más de un proceso maestro, por lo que es erróneo. Se
cierra el programa. \n");
          MPI_Finalize();
          return -1;
      }
      MPI_Comm_spawn(argv[0],  MPI_ARGV_NULL,  PROCESOS,  MPI_INFO_NULL,  0,
MPI_COMM_SELF, &intercom, MPI_ERRCODES_IGNORE);

      // Se envían los datos, primero a pares, luego a los impares.
      // Comunicación punto a punto.
      for (i=0; i<PROCESOS/2; i++) {
          Valor = 1+rand()%100;
          MPI_Send(&Valor, 1, MPI_INT, i*2, 2, intercom);
      }
      for (i=0; i<PROCESOS/2; i++) {
          Valor = 1+rand()%100;
          MPI_Send(&Valor, 1, MPI_INT, i*2+1, 1, intercom);
      }
      // Hacemos un gather para recoger todos los valores
      // numeros = (int *) malloc(PROCESOS*sizeof(int));
      MPI_Gather(&Valor,  1,  MPI_INT,  numeros,  1,  MPI_INT,  MPI_ROOT,
intercom);

      // Ahora recorremos el vector de números y sumamos y
      // restamos para pares e impares
      Suma = 0;
      Resta = 0;
      for (i=0; i<PROCESOS/2; i++) {
          Suma  = Suma + numeros[2*i];
      }
      for (i=0; i<PROCESOS/2; i++) {
          Resta  = Resta - numeros[2*i+1];
      }
      printf("El valor suma de los procesos pares es %d.\n", Suma);
      printf("El valor resta de los procesos impares es %d.\n", Resta);

    } else { //no es el proceso padre, es un hijo
      // Se reciben los valores y se hacen dos intracomunicadores separados
      // para comunicar pares e impares de manera independiente
      MPI_Recv(&Valor,    1,    MPI_INT,    0,    MPI_ANY_TAG,    intercom,
MPI_STATUS_IGNORE);
      Valor = Valor*(rand()%100);
      printf("Valor = %d del proceso %d \n",Valor, miRango);
      if (miRango%2 == 0) color = 0;
      else color=1;
      // dividimos el comunicador de los hijos en dos,
       // le podemos asignar la misma variable
      // pero teniendo en cuenta que los pares tienen uno y los impares
      // otro, y no se pueden comunicar entre ellos con ese comunicador
      MPI_Comm_split(MPI_COMM_WORLD, color, miRango, &ParesImpares);
      MPI_Comm_size(ParesImpares, &procesosdivididos);
      printf("Soy  el  hijo  %d  y  en  mi  grupo  hay  %d  procesos.\n",
miRango,procesosdivididos);
      valores = (int *) malloc(procesosdivididos*sizeof(int));
      //Todos los procesos guardan los valores de su grupo
      MPI_Allgather(&Valor, 1, MPI_INT, valores, 1, MPI_INT, ParesImpares);
```

```
        Valor = 0;
        //sumamos los valores que ha recogido cada proceso
        for (i=0; i<procesosdivididos; i++) {
            Valor = Valor+valores[i];
        }
        printf("Valor final = %d, proceso %d \n",Valor, miRango);
        // Hacemos un gather para recoger todos los valores
        MPI_Gather(&Valor, 1, MPI_INT, numeros, 1, MPI_INT, 0, intercom);

    } //fin del if para distinguir el proceso original

    MPI_Finalize();
    return 0;

} //Cerrar main
```

En los ejemplos anteriores hemos visto como el proceso padre crea un intercomunicador con todos los hijos, y como los hijos pueden crear intracomunicadores entre ellos. Ahora vamos a ver cómo podemos crear, aparte del intercomunicador creado con ***MPI_Comm_spawn***, un intercomunicador para que el proceso padre se comunique sólo con los hijos pares, y otro intercomunicador para que se comunique con los impares. Para ello hay varias alternativas que veremos, para mostrar que raramente hay una solución única gracias a la versatilidad que proporciona MPI.

En el siguiente ejemplo enviaremos números aleatorios a los procesos pares e impares, éstos los multiplicarán por un número aleatorio, y queremos saber el mayor número de los procesos pares y el menor de los impares. Vamos a hacer uso de funciones que no se habían utilizado hasta el momento, como son ***MPI_Intercomm_merge***, que crea un intracomunicador a partir de los grupos de un intercomunicador, para crear un comunicador de iguales (o intracomunicador) entre los líderes de los grupos del nuevo intercomunicador si estos no tenían un intracomunicador común (como ocurre entre un padre y algún hijo creado con ***MPI_Comm_spawn***), por lo que es colectiva sobre todos los procesos del intercomunicador de entrada. Tras esta función, se usará ***MPI_Intercomm_create*** para crear el intercomunicador. De esta manera, con ***MPI_Intercomm_merge*** vamos a crear en nuestro ejemplo un intracomunicador a partir del intercomunicador de padres e hijos, de manera que ahora padre e hijos están en un intracomunicador común. Como hemos dicho, esto tiene la finalidad de crear un intracomunicador común al que pertenezcan los procesos líderes de los grupos del nuevo(s) intercomunicador(es) a crear, es decir un intracomunicador entre padre y proceso 0 de hijos (para los pares), y otro intracomunicador entre proceso padre y proceso 1 de los hijos (para los impares). Podríamos usar dos intracomunicadores de iguales diferentes, pero por simplicidad creamos el intracomunicador que agrupa padres e hijos y nos sirve para crear los dos intercomunicadores. También se creará con ***MPI_Comm_split*** un intracomunicador para cada grupo de procesos hijos, ya que es necesario también para crear los nuevos intercomunicadores. Tras esto se llamaría a la función ***MPI_Intercomm_create*** en la que se le pasarían, para cada

proceso, el intracomunicador de su grupo local (en los hijos creado con ***MPI_Comm_split*** en el ejemplo), el rango del líder del grupo local, el comunicador de iguales entre los líderes, rango del líder remoto dentro del comunicador de iguales, una etiqueta de seguridad y el intercomunicador de salida. Esta función es colectiva para los procesos del futuro intercomunicador y deben llamarla todos los procesos que van a pertenecer al intercomunicador (en grupo local y remoto). Veamos el código cómo sería:

```
/
*****************************************************************************
 * Fichero: Intercomunicadores3.c
 *
 * Compilar con:    mpicc Intercomunicadores3.c -o inter3
 * Ejecutar con:    mpirun -np 1 Inter3

 * Enunciado:

 * Un proceso maestro genera de manera dinámica un número de procesos.
 * A su vez, este proceso maestro generará números aleatorios y se los
 * enviará a los procesos con rango par, que lo multiplicarán por otro
 * número aleatorio. Luego el maestro creará otros números aleatorios y
 * se los enviará a los procesos de rango impar, que también lo
 * multiplicarán por un número aleatorio. Finalmente el proceso maestro
 * deberá conocer cuál es el mayor valor resultante de las multiplicaciones
 * en cada uno de los procesos de rango par, y el menor valor de entre
 * los procesos de rango impar.
 *
 * NOTA: EL número de procesos hijos se considera par por simplicidad
*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "mpi.h"

 #define PROCESOS 10
/
*****************************************************************************
 * Programa principal
*****************************************************************************/
int main(int argc, char *argv[]) {
    // Declaración de variables
    int procesos;// Número de procesos creados de forma estática, que será 1
    int miRango, myrank;
    MPI_Comm intercom, iguales;
    MPI_Comm Pares, Impares, paresImpares;  // Intercomunicador con los grupos
    time_t t;
    int pares[PROCESOS/2], impares[PROCESOS/2];
    int maximo, minimo, i, valor, color;

    // Inicia MPI
    MPI_Init(&argc, &argv);

    // Obtiene su rango y número de procesos
    MPI_Comm_size(MPI_COMM_WORLD, &procesos);
    MPI_Comm_rank(MPI_COMM_WORLD, &miRango);
```

```
    // Semilla para generar secuencias aleatorias
    srand((unsigned) time(&t));


    MPI_Comm_get_parent(&intercom); // Para conocer si tiene padre o no

    if (intercom == MPI_COMM_NULL) {

       if (procesos>1) {
           printf(" Hay más de un proceso maestro, por lo que es erróneo. Se
cierra el programa. \n");
           MPI_Finalize();
           return -1;
       }
       // Se crean los procesos Padres
       MPI_Comm_spawn(argv[0], MPI_ARGV_NULL, PROCESOS, MPI_INFO_NULL, 0,
MPI_COMM_SELF, &intercom, MPI_ERRCODES_IGNORE);

       // Se crean los números.
       for (i=0; i<PROCESOS/2; i++) {
           pares[i] = 1+rand()%100;
       }
       for (i=0; i<PROCESOS/2; i++) {
           impares[i] = 1+rand()%100;
       }

       // Creamos comunicador de iguales para crear luego
      // los intercomunicadores con los grupos
       MPI_Intercomm_merge(intercom, 0, &iguales);

       // Creamos el intercomunicador para los pares en el lado del padre
       MPI_Intercomm_create(MPI_COMM_SELF, 0, iguales, 1, 2, &Pares);

       // Creamos el intercomunicador para los impares en el lado del padre
       MPI_Intercomm_create(MPI_COMM_SELF, 0, iguales, 2, 1, &Impares);

       // Se envían los datos a los pares e impares con un scatter
       // usando el intercomunicador dedicado a cada grupo
       MPI_Scatter(pares, 1, MPI_INT, &valor, 1, MPI_INT, MPI_ROOT, Pares);
       MPI_Scatter(impares, 1, MPI_INT, &valor, 1, MPI_INT, MPI_ROOT,
Impares);

       // Ahora con dos Reduce ya sabremos cuáles son los máximo y mínimo
       MPI_Reduce(0, &maximo, 1, MPI_INT, MPI_MAX, MPI_ROOT, Pares);
       MPI_Reduce(0, &minimo, 1, MPI_INT, MPI_MIN, MPI_ROOT, Impares);

       printf("El valor máximo de los procesos pares es %d.\n", maximo);
       printf("El valor mínimo de los procesos impares es %d.\n", minimo);

    } else { //no es el proceso padre, es un hijo
       // Comunicador de iguales para crear el intercomunicador
       MPI_Intercomm_merge(intercom, 1, &iguales);
       // Dividimos el grupo local en dos intracomunicadores diferentes
       // aunque con la misma variable, para crear los intercomunicadores
       // con el proceso padre
       color = miRango%2;
       MPI_Comm_split(MPI_COMM_WORLD, color, miRango, &paresImpares);
       // Creamos los intercomunicadores para cada grupo de procesos
       // Podría haberse usado la misma variable para ambos intercomunicadores
       // en los hijos y se hubieran ahorrado líneas de código, pero se ha
```

```
       // separado para mostrar claramente como hay dos intercomunicadores
       if (color == 0) { //pares
           MPI_Intercomm_create(paresImpares, 0, iguales, 0, 2, &Pares);
           MPI_Scatter(pares, 1, MPI_INT, &valor, 1, MPI_INT, 0, Pares);
           // Obtengo el rango local en el nuevo intercomunicador
           MPI_Comm_rank(Pares, &myrank);
           printf("Valor = %d del proceso con rango %d en MPI_COMM_WORLD y
rango %d en Pares \n",valor, miRango, myrank);
           valor = valor*(rand()%100);
           MPI_Reduce(&valor, 0, 1, MPI_INT, MPI_MAX, 0, Pares);
       } else {
           MPI_Intercomm_create(paresImpares, 0, iguales, 0, 1, &Impares);
           MPI_Scatter(impares, 1, MPI_INT, &valor, 1, MPI_INT, 0, Impares);
           // Obtengo el rango local en el nuevo intercomunicador
           MPI_Comm_rank(Impares, &myrank);
           printf("Valor = %d del proceso con rango %d en MPI_COMM_WORLD y
rango %d en Impares \n",valor, miRango, myrank);
           valor = valor*(rand()%100);
           MPI_Reduce(&valor, 0, 1, MPI_INT, MPI_MIN, 0, Impares);
       }

    } //fin del if para distinguir el proceso original

    MPI_Finalize();
    return 0;

} //Cerrar main
```

En el siguiente ejemplo, se realiza lo mismo pero la creación de los intercomunicadores se realiza con las funciones ***MPI_Comm_group*** y ***MPI_Comm_create*** en lugar de ***MPI_Intercomm_create***, aunque a efectos prácticos el resultado es el mismo.

```
/
*****************************************************************************
 * Fichero: Intercomunicadores4.c
 *
 * Compilar con:    mpicc Intercomunicadores4.c -o inter4
 * Ejecutar con:    mpirun -np 1 inter4

 * Un proceso maestro genera de manera dinámica M procesos, siendo M un número
 * par. A su vez, este proceso maestro generará M/2 números aleatorios y se
 * los enviará a los procesos con rango par, que lo multiplicarán por otro
 * número aleatorio. Luego el maestro creará otros M/2 números aleatorios y se
 * los enviará a los procesos de rango impar, que también lo multiplicarán por
 * un número aleatorio. Finalmente el proceso maestro deberá conocer cuál es
 * el mayor valor resultante de las multiplicaciones en cada uno de los
 * procesos de rango par, y el menor valor de entre los procesos de rango
 * impar.
*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "mpi.h"

#define PROCESOS 6
```

```
/
*****************************************************************************
 * Programa principal
*****************************************************************************/
int main(int argc, char *argv[]) {
    // Declaración de variables
    int procesos;
    int miRango;
    MPI_Comm intercom, intercom2;
    MPI_Comm ParesImpares, ParesImpares2, Pares, Impares;
    time_t t;
    int *pares, *impares;
    int rangos[PROCESOS/2], rangos2[PROCESOS/2];;
    int maximo, minimo, i;
    int rank;
    int valor;
    int gr_local, gr_remoto;

    MPI_Group grupo_padre, grupo_hijos, grupo_hijos_separados,
grupo_hijos_separados2;

    pares = (int*) malloc(sizeof(int)*PROCESOS);
    impares = (int*) malloc(sizeof(int)*PROCESOS);

    // Inicia MPI
    MPI_Init(&argc, &argv);

    // Obtiene su rango y número de procesos
    MPI_Comm_size(MPI_COMM_WORLD, &procesos);
    MPI_Comm_rank(MPI_COMM_WORLD, &miRango);
    valor = 0;

    // Semilla para generar secuencias aleatorias
    srand((unsigned) time(&t));


    MPI_Comm_get_parent(&intercom); // Para conocer si tiene padre o no

    if ((procesos>1)&&(intercom == MPI_COMM_NULL)) {
      printf(" Hay más de un proceso maestro, por lo que es erróneo. Se
cierra el programa. \n");
      MPI_Finalize();
      return -1;
    }

    //printf( " %d  %d \n", sizeof (long int), sizeof (MPI_INT));
    if (intercom == MPI_COMM_NULL) {
      // Se crean los procesos hijos
      MPI_Comm_spawn(argv[0], MPI_ARGV_NULL, PROCESOS, MPI_INFO_NULL, 0,
MPI_COMM_SELF, &intercom, MPI_ERRCODES_IGNORE);

      // Se crean los números.
      for (i=0; i<PROCESOS/2; i++) {
          pares[i] = i*2;
          printf("Pares %d \n", pares[i]);
      }
      for (i=0; i<PROCESOS/2; i++) {
          impares[i] = i*2+1;
          printf("Impares %d \n", impares[i]);
      }
```

```
        // duplicamos el comunicador intercom por claridad a la hora
        // de crear los dos intercomunicadores con los pares e impares
        // es colectiva, por lo que los hijos deben llamarla también
        MPI_Comm_dup(intercom, &intercom2);

        //Creamos un grupo local con el padre a partir del intercomunicador
        MPI_Comm_group(intercom, &grupo_padre);
        MPI_Group_size(grupo_padre, &rank);

        // Creamos el intercomunicador para los pares, de esta manera
        // aquí estamos determinando el grupo local del intercomunicador
        // de los procesos pares, los hijos harán el otro grupo.
        MPI_Comm_create(intercom, grupo_padre, &Pares);
        // Liberamos el grupo
        MPI_Group_free(&grupo_padre);

        // ahora podemos comprobar el tamaño del grupo local y remoto
        MPI_Comm_size(Pares, &gr_local);
        MPI_Comm_remote_size(Pares, &gr_remoto);
        printf("Soy el padre, en el intercomunicador de Pares somos %d
procesos en el grupo local y hay %d procesos en el remoto \n", gr_local,
gr_remoto);

        // Creamos el intercomunicador para los impares de la misma manera
        // pero usando el intercomunicador duplicado, no es necesario
        // y podría hacerse sobre el mismo

        // Primero el grupo local del nuevo intercomunicador con el padre
        MPI_Comm_group(intercom2, &grupo_padre);
        // Creamos el intercomunicador en el lado del padre
        MPI_Comm_create(intercom2, grupo_padre, &Impares);
        // liberamos el grupo padre
        MPI_Group_free(&grupo_padre);

        MPI_Comm_size(Impares, &gr_local);
        MPI_Comm_remote_size(Impares, &gr_remoto);
        printf("Soy el padre, en el intercomunicador de Impares somos %d
procesos en el grupo local y hay %d procesos en el remoto \n", gr_local,
gr_remoto);

        // Se envían los datos a los pares e impares con un scatter
        MPI_Scatter(pares, 1, MPI_INT, &valor, 1, MPI_INT, MPI_ROOT, Pares);
        MPI_Scatter(impares, 1, MPI_INT, &valor, 1, MPI_INT, MPI_ROOT,
Impares);

        // Ahora con dos Reduce ya sabremos cuáles son los máximo y mínimo
        MPI_Reduce(&valor, &maximo, 1, MPI_INT, MPI_MAX, MPI_ROOT, Pares);
        MPI_Reduce(0, &minimo, 1, MPI_INT, MPI_MIN, MPI_ROOT, Impares);

        printf("El valor máximo de los procesos pares es %d.\n",maximo);
        printf("El valor mínimo de los procesos impares es %d.\n", minimo);
        MPI_Comm_free(&Pares);
        MPI_Comm_free(&Impares);

    } else { //no es el proceso padre, es un hijo
        // Duplicamos el intercomunicador para crear los intercomunicadores,
        // pero es por claridad, no sería necesario
        MPI_Comm_dup(intercom, &intercom2);
```

```
       // Creamos los grupos indicando los rangos de los procesos que
       // pertenecerán a cada uno
       for (i=0; i<PROCESOS/2; i++) {
           rangos[i] = i*2; // rangos pares
           rangos2[i] = i*2+1; // rangos impares
       }
       // Ahora creamos los grupos separados dependiendo de si
      // el proceso es par o impar

       // Creamos el grupo de hijos completo
       MPI_Comm_group(intercom,&grupo_hijos);
       // Creamos el grupo de los pares
       MPI_Group_incl(grupo_hijos, PROCESOS/2, rangos,
&grupo_hijos_separados);
       MPI_Group_free(&grupo_hijos);
       // Creamos el intercomunicador de los pares, se ha cambiado su variable
       // para remarcar que no tiene que ser la misma que en el padre
       // y que el creado en el padre aún no existe en los hijos hasta crearlo
       // en este lado del comunicador
       MPI_Comm_create(intercom, grupo_hijos_separados, &ParesImpares);
       MPI_Group_free(&grupo_hijos_separados);

       // Repetimos para los impares con el intercomunicador duplicado

       // Creamos el grupo de hijos completo
       MPI_Comm_group(intercom2,&grupo_hijos);
       // Creamos el grupo de impares
       MPI_Group_incl(grupo_hijos, PROCESOS/2, rangos2,
&grupo_hijos_separados2);
       MPI_Group_free(&grupo_hijos);
       // Creamos el intercomunicador de los impares
       MPI_Comm_create(intercom2, grupo_hijos_separados2, &ParesImpares2);
       MPI_Group_free(&grupo_hijos_separados2);

       // Recibimos los valores del scatter correspondiente
       // Si hubieramos usado la misma variable para los comunicadores
       // de pares e impares, el siguiente if sería innecesario ya que
       // aunque tuvieran la misma variable, serían diferentes y cada uno
       // llamaría al suyo, pero se han puesto diferentes y se
       // ha duplicado el código para clarificarlo. Podéis probar a
       // simplificarlo usando los mismos y ver que se obtiene lo mismo
       if (miRango%2==0) { //par
           MPI_Comm_rank(ParesImpares, &rank);
           printf("Soy el hijo %d y en el nuevo comunicador soy %d \n",
miRango, rank);
           // Se recibe el valor con Scatter
           MPI_Scatter(0, 0, MPI_INT, &valor, 1, MPI_INT, 0, ParesImpares);
           printf("Soy el hijo %d y he recibido %d \n", miRango, valor);
           valor = valor*(rand()%100);
           // Se envía al maestro con Reduce
           MPI_Reduce(&valor, &maximo, 1, MPI_INT, MPI_MAX, 0, ParesImpares);
           MPI_Comm_free(&ParesImpares);
       }      else {
           MPI_Comm_rank(ParesImpares2, &rank);
           printf("Soy el hijo %d y en el nuevo comunicador soy %d \n",
miRango, rank);
           // Se recibe el valor con Scatter
           MPI_Scatter(0, 0, MPI_INT, &valor, 1, MPI_INT, 0, ParesImpares2);
           printf("Soy el hijo %d y he recibido %d \n", miRango, valor);
           valor = valor*(rand()%100);
```

```
            // Se envía al maestro con Reduce
            MPI_Reduce(&valor, 0, 1, MPI_INT, MPI_MAX, 0, ParesImpares2);
            MPI_Comm_free(&ParesImpares2);
        }

    } //fin del if para distinguir el proceso original
    MPI_Comm_free(&intercom);
    MPI_Comm_free(&intercom2);
    MPI_Finalize();
    return 0;

} //Cerrar main
```

Finalmente en el siguiente ejemplo podemos ver el mismo ejemplo pero creando los intercomunicadores con la función ***MPI_Comm_split*** viendo de esta manera las diferentes posibilidades y versatilidad de MPI para manejar grupos y comunicadores.

```
/
*****************************************************************************
 * Fichero: Intercomunicadores5.c
 *
 * Compilar con:    mpicc Intercomunicadores5.c -o inter5
 * Ejecutar con:    mpirun -np 1 inter5

 * Un proceso maestro genera de manera dinámica M procesos, siendo M un número
 * par. A su vez, este proceso maestro generará M/2 números aleatorios y se
 * los enviará a los procesos con rango par, que lo multiplicarán por otro
 * número aleatorio. Luego el maestro creará otros M/2 números aleatorios y se
 * los enviará a los procesos de rango impar, que también lo multiplicarán por
 * un número aleatorio. Finalmente el proceso maestro deberá conocer cuál es
 * el mayor valor resultante de las multiplicaciones en cada uno de los
 * procesos de rango par, y el menor valor de entre los procesos de rango
 * impar.
*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "mpi.h"

#define PROCESOS 10
/
*****************************************************************************
 * Programa principal: Creación de intercomunicadores con Split
*****************************************************************************/
int main(int argc, char *argv[]) {
    // Declaración de variables
    int procesos;
    int miRango;
    MPI_Comm intercom, intercom2;
    MPI_Comm Pares, Impares;
    time_t t;
    int pares[PROCESOS/2], impares[PROCESOS/2];
    int maximo, minimo, i, valor, color;

    // Inicialización de MPI
```

```
    MPI_Init(&argc, &argv);

    // Obtiene su rango y número de procesos
    MPI_Comm_size(MPI_COMM_WORLD, &procesos);
    MPI_Comm_rank(MPI_COMM_WORLD, &miRango);

    // Semilla para generar secuencias aleatorias
    srand((unsigned) time(&t));


    MPI_Comm_get_parent(&intercom); // Para conocer si tiene padre o no

    if (intercom == MPI_COMM_NULL) {
       if (procesos>1) {
           printf(" Hay más de un proceso maestro, por lo que es erróneo. Se
cierra el programa. \n");
           MPI_Finalize();
           return -1;
       }
       // Se crean los procesos hijos
       MPI_Comm_spawn(argv[0], MPI_ARGV_NULL, PROCESOS, MPI_INFO_NULL, 0,
MPI_COMM_SELF, &intercom, MPI_ERRCODES_IGNORE);

       // Se crean los números.
       for (i=0; i<PROCESOS/2; i++) {
           pares[i] = 1+rand()%100;
       }
       for (i=0; i<PROCESOS/2; i++) {
           impares[i] = 1+rand()%100;
       }
       // Duplicamos el intercomunicador por claridad en la creación
       // de los nuevos intercomunicadores
       MPI_Comm_dup(intercom, &intercom2);

       // Creamos comunicadores, el primero para los pares y el segundo
       // para los impares. Son dos llamadas a split por lo que los hijos
       // deben hacer también dos llamadas a split, uno con color en su
       // comunicador deseado de salida y la otra con MPI_UNDEFINED. Estas dos
       // llamadas son necesarias ya que para crear el intercomunicador el
       // maestro debe tener el mismo color que los hijos, por lo que para
       // crear los dos debe llamar a split con ambos colores.
       MPI_Comm_split(intercom, 0, miRango, &Pares);
       MPI_Comm_split(intercom2, 1, miRango, &Impares);

       // Se envían los datos a los pares e impares con un scatter en
       // sus correspondientes intercomunicadores
       MPI_Scatter(pares, 1, MPI_INT, &valor, 1, MPI_INT, MPI_ROOT, Pares);
       MPI_Scatter(impares, 1, MPI_INT, &valor, 1, MPI_INT, MPI_ROOT,
Impares);

       // Ahora con dos Reduce ya sabremos cuáles son los máximo y mínimo
       MPI_Reduce(0, &maximo, 1, MPI_INT, MPI_MAX, MPI_ROOT, Pares);
       MPI_Reduce(0, &minimo, 1, MPI_INT, MPI_MIN, MPI_ROOT, Impares);

       printf("El valor máximo de los procesos pares es %d.\n", maximo);
       printf("El valor mínimo de los procesos impares es %d.\n", minimo);

    } else { //no es el proceso original, es un hijo
       // Con color distinguimos entre pares e impares
       color = miRango%2;
```

```
        // Al igual que en el maestro duplicamos el intercomunicador
        // para que todos los procesos involucrados lo tengan
        MPI_Comm_dup(intercom, &intercom2);
        // Creamos los intercomunicadores
        if (color == 0) { //pares
            // Los pares crean su comunicador de pares
            MPI_Comm_split(intercom, color, miRango, &Pares);
            // y llaman al split para crear el comunicador de impares
            // con color = MPI_UNDEFINED ya que para esta llamada no
            // deseamos que los pares creen comunicador alguno.
            MPI_Comm_split(intercom2, MPI_UNDEFINED, miRango, &Impares);
            // Recibe los valores del padre
            MPI_Scatter(pares, 1, MPI_INT, &valor, 1, MPI_INT, 0, Pares);
            valor = valor*(rand()%100);
            // Se devuelve para calcular el máximo
            MPI_Reduce(&valor, 0, 1, MPI_INT, MPI_MAX, 0, Pares);
        } else {
            // Los impares llaman a split con MPI_UNDEFINED en color
            // para el comunicador de los pares
            MPI_Comm_split(intercom, MPI_UNDEFINED, miRango, &Pares);
            // Y después crean su intercomunicador de impares
            MPI_Comm_split(intercom2, color, miRango, &Impares);
            // Se reciben los datos
            MPI_Scatter(impares, 1, MPI_INT, &valor, 1, MPI_INT, 0, Impares);
            valor = valor*(rand()%100);
            // Se devuelven al padre para calcular el mínimo
            MPI_Reduce(&valor, 0, 1, MPI_INT, MPI_MIN, 0, Impares);
        }
    } //fin del if para distinguir el proceso original

    MPI_Finalize();
    return 0;

} //Cerrar main
```

Es importante fijarse en los últimos ejemplos como para cada llamada debe utilizarse el comunicador adecuado, así en el último ejemplo dependiendo de si estamos creando el intercomunicador **Pares** o **Impares**, se usara el **intercom** o **intercom2**, ya que en caso de no usar el comunicador adecuado, el programa no funcionará.

De este modo hemos visto como tenemos múltiples posibilidades para abordar un mismo problema.