# 2.2 Paradigmas de programación paralela: paso de mensajes

*Computación de Altas Prestaciones*

Grado en Ingeniería de Computadores

Raúl Durán Díaz

Curso académico 2018–2019

## Índice

# 1. Funciones básicas en MPI

### Sistemas de memoria distribuida

- Desde el punto de vista hardware, estas máquinas:
  - Están típicamente construidas por medio de computadores completos (procesador + memoria), incluyendo E/S.
  - Interconexión entre ellos por medio de redes (estáticas o dinámicas).
  - Comunicación por medio de operaciones explícitas de E/S.
- Desde el punto de vista del modelo de programación:
  - Acceso directo sólo a direcciones privadas (memoria local).
  - Comunicación por intercambio de mensajes.
- Existe intervención apreciable del sistema operativo.
- Normalmente se programa por intermedio de librerías.

### MPI

- MPI significa **Message Passing Interface**.
- Es un conjunto de funciones en **C** (o subrutinas en **FORTRAN**) con las que se puede implementar un programa usando paso de mensajes.
- MPI permite coordinar la ejecución del programa en múltiples procesadores con memoria distribuida.
- MPI está normalizada:
  - Cualquier programa escrito con esta librería funcionará sobre cualquier máquina en que MPI esté instalado.

### Funciones básicas en MPI

- En el comienzo de todo programa debemos especificar:

  ```
  # include "mpi.h"
  ```

  - Con ello, disponemos de las funciones y constantes definidas en la librería.
- Todas las funciones y constantes de MPI comienzan con el prefijo `MPI_`.
  - En el caso de las funciones sigue una letra mayúscula y las demás minúsculas: `MPI_Init`.
  - En el caso de constantes, son todas mayúsculas: `MPI_CHAR`.
- Todas las funciones devuelven un entero `int` salvo que se diga lo contrario.

### Funciones básicas en MPI

- Antes de comenzar a usar la librería debemos llamar a la función `MPI_Init`.
  - Sólo debe hacerse una vez.
- Para terminar, debe llamarse a la `MPI_Finalize`.

```
# include "mpi.h"
...
main(int argc, char *argv[]) {
   ...
   /* Antes de este punto, no llamar a ninguna función MPI */
   MPI_Init(&argc, &argv);
   ...
   MPI_Finalize();
   /* Pasado este punto, no llamar a ninguna función MPI */
   ...
}
```

### Funciones básicas en MPI

- Llamamos grupo de comunicación (*communicator*) a una familia de procesos que tienen permitido el intercambio de mensajes.
- Existe un grupo de comunicación por defecto, llamado `MPI_COMM_WORLD`. En él se encuentran todos los procesos en el momento de arranque.
- Cada proceso se identifica por su *rango* (un identificador numérico) que se puede obtener con `MPI_Comm_rank`.
- El número total de procesos en un grupo de comunicación se puede obtener con `MPI_Comm_size`.

### Funciones básicas en MPI

- Para enviar y recibir mensajes, utilizamos las funciones básicas:
  - `MPI_Send`
  - `MPI_Recv`
- Los mensajes pueden sólo intercambiarse dentro del mismo grupo de comunicación.
- Cada mensaje enviado lleva una etiqueta identificadora (*tag*) que es única dentro de cada grupo de comunicación.

### Soporte del sistema

- Antes de poder ejecutar el ejemplo, cada máquina debe tener acceso a una copia.
  - O bien, todas ellas tienen acceso a la misma copia (disco compartido).
- ¿Qué ocurre cuando lanzamos una ejecución?
  - Cada máquina comienza la ejecución de su copia.
  - Cada máquina realiza una ejecución independiente.
- Cada proceso puede ejecutar distintas zonas del programa si la lógica de éste depende de la identificación (rango) del proceso.

### Envío de mensajes

- La función `MPI_Send` tiene el siguiente prototipo:

```
MPI_Send(void         *mensaje,    /* <== */
         int           cuenta,     /* <== */
         MPI_Datatype  tipodato,   /* <== */
         int           destino,    /* <== */
         int           etiqueta,   /* <== */
         MPI_Comm      grupo_com); /* <== */
```

- Los tipos de datos son, por ejemplo:

  `MPI_CHAR`
  `MPI_SHORT`
  `MPI_FLOAT`
  `....`

### Recepción de mensajes

- La función `MPI_Recv` tiene el siguiente prototipo:

```
MPI_Recv(void         *mensaje,   /* ==> */
         int           cuenta,    /* <== */
         MPI_Datatype  tipodato,  /* <== */
         int           remitente, /* <== */
         int           etiqueta,  /* <== */
         MPI_Comm      grupo_com, /* <== */
         MPI_Status   *status);   /* ==> */
```

- El parámetro cuenta especifica el número máximo de elementos de tipo `tipodato` que caben en el buffer mensaje.
  - Si se envía un mensaje más largo, se produce un error de desbordamiento.

### Recepción de mensajes

- La variable de tipo `MPI_Status` tiene, al menos, los siguientes campos:

  `MPI_SOURCE`
  `MPI_TAG`
  `MPI_ERROR`

- También contiene información acerca del tamaño del mensaje recibido, que se obtiene mediante la función:

```
MPI_Get_count(MPI_Status   *status,   /* <== */
              MPI_Datatype  datatype, /* <== */
              int          *cuenta);  /* ==> */
```

  Atención: cuenta obtendrá el número de elementos de tipo `datatype` (y no el número de bytes).

**Funciones auxiliares**

- Otras dos funciones muy usadas:

```
MPI_Comm_rank(MPI_Comm comunicador, /* <== */
              int      *rango);     /* ==> */
```

  - Devuelve al proceso que la llama (en la variable `rango`) su rango dentro del comunicador `comunicador`.
  - El comunicador es una estructura que, dicho de modo sencillo, agrupa los procesos que pueden intercambiar mensajes.

```
MPI_Comm_size(MPI_Comm  comunicador, /* <== */
              int      *tamanyo);    /* ==> */
```

  - Devuelve en la variable `tamanyo` el número de procesos enganchados al comunicador identificado como `comunicador`.

**Envío y recepción integradas**

A veces viene bien combinar en una llamada una recepción de un proceso junto con un envío a otro. Para ello tenemos la función `MPI_Sendrecv`.

```
MPI_Sendrecv(void         *buff_envio,    /* <== */
             int           cuentaenvio,   /* <== */
             MPI_Datatype  tipodatoenvio, /* <== */
             int           destino,       /* <== */
             int           etiquetaenvio, /* <== */
             void         *buff_recep,    /* ==> */
             int           cuentarecep,   /* <== */
             MPI_Datatype  tipodatorecep, /* <== */
             int           remitente,     /* <== */
             int           etiquetarecep, /* <== */
             MPI_Comm      comunicador,   /* <== */
             MPI_Status   *status);       /* ==> */
```

**Envío y recepción integradas**

Para algunas aplicaciones viene mejor todavía que el buffer de recepción y el de envío sea el mismo. Esta función garantiza que se envía primero y se recibe después. El tipo y número de datos transmitidos ha de ser idéntico para envío y recepción.

```
MPI_Sendrecv_replace(void         *buff_enviorecep, /* <=> */
                     int           cuenta,          /* <== */
                     MPI_Datatype  tipodato,        /* <== */
                     int           destino,         /* <== */
                     int           etiquetaenvio,   /* <== */
                     int           remitente,       /* <== */
                     int           etiquetarecep,   /* <== */
                     MPI_Comm      comunicador,     /* <== */
                     MPI_Status   *status);         /* ==> */
```

**Comunicación no bloqueante**

- Puede ser interesante a veces no esperar la terminación de una operación de envío o recepción, si la lógica de nuestro algoritmo lo permite. De esa manera, podemos *solapar* computación y comunicación.
- Para ello, se necesita que el hardware lo permita (¡no siempre es posible!).

**Comunicación no bloqueante**

- MPI proporciona unas rutinas de envío y recepción que son *no bloqueantes*: solamente inician la función y regresan antes de completarse. Estas rutinas son `MPI_Isend` y `MPI_Irecv`.
- Además, tenemos una rutina para comprobar si el envío o recepción ha terminado, y otra para esperar la terminación de cualquiera de ellas.

**Envío no bloqueante**

Los argumentos del envío no bloqueante son similares a su compañero bloqueante, con la aparición de un nuevo argumento, en donde la función devuelve una información con la que posteriormente se puede gestionar este envío.

```
MPI_Isend(void        *mensaje,    /* <== */
          int          cuenta,     /* <== */
          MPI_Datatype tipodato,   /* <== */
          int          destino,    /* <== */
          int          etiqueta,   /* <== */
          MPI_Comm     grupo_com,  /* <== */
          MPI_Request  *req);      /* ==> */
```

**Recepción no bloqueante**

Los argumentos de la recepción no bloqueante también son similares a su compañera bloqueante. El argumento del estado se sustituye por un nuevo argumento con el que se gestiona posteriormente el estado de la recepción.

```
MPI_Irecv(void        *mensaje,    /* ==> */
          int          cuenta,     /* <== */
          MPI_Datatype tipodato,   /* <== */
          int          remitente,  /* <== */
          int          etiqueta,   /* <== */
          MPI_Comm     grupo_com,  /* <== */
          MPI_Request  *req);      /* ==> */
```

**Gestión de los envíos/recepciones**

- Para gestionar los envíos/recepciones pendientes, disponemos de dos funciones:
  - `MPI_Test`
    Permite comprobar si la correspondiente operación ha terminado ya o no.
  - `MPI_Wait`
    Permite realizar una espera hasta la terminación de la correspondiente operación.
- En ambos casos, la operación se identifica mediante el objeto de tipo `MPI_Request` asociado a ella en el momento de solicitarla.

**Función para comprobar el estado de una operación pendiente**

Disponemos para ello de la función siguiente:

```
MPI_Test(MPI_Request  *req,     /* <== */
         int          *flag,    /* ==> */
         MPI_Status   *status); /* ==> */
```

El primer parámetro, `req`, identifica la operación cuyo estado queremos comprobar. El resultado aparece en la variable `flag`, que si toma valor de `TRUE` (es decir, no nulo), indica la terminación de la operación, con lo que la variable `status` adquiere el mismo significado que en la recepción ordinaria; y toma `FALSE` en caso contrario (en este caso `status` no tiene sentido).

**Función para esperar la terminación de una operación pendiente**

Disponemos para ello de la función siguiente:

```
MPI_Wait(MPI_Request  *req,     /* <== */
         MPI_Status   *status); /* ==> */
```

En este caso, la función queda bloqueada hasta que la operación identificada en `req` se termina. La variable `status` recibe información sobre la terminación de la operación.

# 2. Comunicación colectiva

### Operación de difusión

- La siguiente función difunde un mensaje a todos los procesos asociados a un comunicador:

```
MPI_Bcast(void         *mensaje,     /* <==> */
          int           cuenta,      /* <==  */
          MPI_Datatype  tipodato,    /* <==  */
          int           raiz,        /* <==  */
          MPI_Comm      grupo_com);  /* <==  */
```

### Operación de difusión

- La llamada a la función `MPI_Bcast` puede resultar en
  - recepción de datos, si el rango del proceso es distinto de `raiz`.
  - envío de datos, si el rango del proceso es igual a `raiz`.
- El valor de `tipodato` y `cuenta` ha de ser igual para todos. Los procesos afectados serán todos los que se encuentren en el grupo de comunicaciones `grupo_com`.
- El sistema MPI garantiza que si un proceso difunde varios mensajes (es decir, realiza varias llamadas a `MPI_Bcast`), éstos serán recibidos por los demás procesos en el mismo orden en que fueron emitidos.

### Ejemplo de reducción

- La integración con división en trapecios tiene dos fases:
  - Cálculo de las integrales parciales de cada grupo de trapecios.
  - Suma de los resultados.
- La primera fase puede fácilmente distribuirse de modo equilibrado entre los distintos procesos.
- En cambio, la fase de suma la realiza exclusivamente un proceso (por ejemplo, el proceso 0).
  - *Sería interesante redistribuir el trabajo de suma entre los distintos procesos para equilibrar este trabajo.*

### Operación de reducción

- Para resolver este problema, MPI proporciona la función `MPI_Reduce` con el siguiente prototipo:

```
MPI_Reduce(void         *operando,   /* <== */
           void         *resultado,  /* ==> */
           int           cuenta,     /* <== */
           MPI_Datatype  tipodato,   /* <== */
           MPI_Op        operacion,  /* <== */
           int           raiz,       /* <== */
           MPI_Comm      grupo_com); /* <== */
```

### Operación de reducción

- `MPI_Reduce` ha de ser llamado en todos los procesos del grupo de comunicación `grupo_com` y `cuenta`, `tipodato` y `operacion` han de valer lo mismo en todos los procesos. El argumento operacion puede valer

  `MPI_SUM`
  `MPI_PROD`
  `MPI_MAX`
  `MPI_MIN`
  `...`

- Obsérvese que la variable resultado sólo tiene sentido en el proceso `raiz`. Aun así, los demás procesos también han de especificarla.

### Operación de reducción

- `MPI_Reduce` combina los operandos almacenados en operando usando la operación operacion y almacena el resultado en resultado en el proceso `raiz`.
- Tanto `operando` como `resultado` se refieren a `cuenta` elementos de tipo `tipodato`.

### Operación de reducción total

- En algunos casos nos interesa que la reducción se efectúe en todos los procesos. Para ello existe la función `MPI_Allreduce` cuyo prototipo es:

```
MPI_Allreduce(void         *operando,     /* <== */
              void         *resultado,    /* ==> */
              int           cuenta,       /* <== */
              MPI_Datatype  tipodato,     /* <== */
              MPI_Op        operacion,    /* <== */
              MPI_Comm      grupo_com);   /* <== */
```

- Se usa exactamente igual que `MPI_Reduce`, pero el resultado de la reducción se acumula en resultado en todos los procesos pertenecientes al grupo de comunicación `grupo_com`. Por ello no es necesario el parámetro `raiz`, como en el otro caso.

### Barreras

- Las barreras están directamente implementadas en MPI:

```
MPI_Barrier(MPI_Comm grupo_com); /* <== */
```

- Cada proceso del grupo de comunicación `grupo_com` se quedará bloqueado hasta que todos ellos hayan llamado a esta función.
- Cuando el último de ellos la llame, todos se desbloquean simultáneamente.
- Es un mecanismo de sincronización.

### Operación de reunión

- Otras funciones de comunicación colectiva son las siguientes:

```
MPI_Gather(void         *buffer_envio,     /* <== */
           int           cuenta_envio,     /* <== */
           MPI_Datatype  tipo_envio,       /* <== */
           void         *buffer_recepcion, /* ==> */
           int           cuenta_recepcion, /* <== */
           MPI_Datatype  tipo_recepcion,   /* <== */
           int           raiz,             /* <== */
           MPI_Comm      grupo_com);       /* <== */
```

- Cada proceso del grupo de comunicación `grupo_com` envía los contenidos de `buffer_envio` al proceso `raiz`.

### Operación de reunión

- El proceso `raiz` concatena los datos recibidos por orden de rango en `buffer_recepcion`, es decir, los datos del proceso 0, a continuación los del 1, etc.
- Los argumentos de recepción sólo son significativos en el proceso `raiz`.
- El argumento `cuenta_recepcion` indica el número de items recibidos de cada proceso (no el total).

### Operación de dispersión

```
MPI_Scatter(void         *buffer_envio,     /* <== */
            int           cuenta_envio,     /* <== */
            MPI_Datatype  tipo_envio,       /* <== */
            void         *buffer_recepcion, /* ==> */
            int           cuenta_recepcion, /* <== */
            MPI_Datatype  tipo_recepcion,   /* <== */
            int           raiz,             /* <== */
            MPI_Comm      grupo_com);       /* <== */
```

- El proceso con rango `raiz` distribuye los contenidos del buffer `buffer_envio` en tantos segmentos como procesos haya, cada uno con un tamaño de `cuenta_envio` items.
- Los argumentos de envío son significativos sólo en el proceso `raiz`.

### Operación de reunión total

```
MPI_Allgather(void         *buffer_envio,     /* <== */
              int           cuenta_envio,     /* <== */
              MPI_Datatype  tipo_envio,       /* <== */
              void         *buffer_recepcion, /* ==> */
              int           cuenta_recepcion, /* <== */
              MPI_Datatype  tipo_recepcion,   /* <== */
              MPI_Comm      grupo_com);       /* <== */
```

- Cada proceso del grupo de comunicación `grupo_com` envía los contenidos de `buffer_envio` al todos los demás procesos.
- El efecto es equivalente a llamar a `MPI_Gather` tantas veces como procesos haya, actuando sucesivamente cada uno de ellos como `raiz`.

# 3. Agrupación de datos

### Vista general

- Enviar mensajes es una operación costosa.
- Hay que tratar de enviar el mínimo número posible.
- La solución obvia es agrupar los mensajes.
- Existen tres mecanismos para agrupar mensajes:
  - Parámetro `cuenta`.
  - Tipos de datos derivados.
  - Las rutinas `MPI_Pack`/`MPI_Unpack`.

### Parámetro `cuenta`

- Recordemos que las funciones `MPI_Send`, `MPI_Receive`, `MPI_Bcast`, `MPI_Reduce` tienen todas el parámetro `cuenta` y el parámetro `tipodato`.
- Con ellos, el usuario puede agrupar datos que sean del mismo tipo básico en un solo mensaje.
- Condición imprescindible:
  - los datos han de estar almacenados en memoria contigua.
- Esto es útil para los vectores o matrices.

### Parámetro `cuenta`

*Ejemplo* 1. Enviar la segunda mitad de un vector de 100 flotantes desde el proceso 0 al 1.

```
float vector[100];
MPI_Status status;
int p;
int mi_rango;
...
/* Inicializar vector y enviar */
   if (mi_rango == 0) {
   ...
      MPI_Send(vector+50, 50, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
   } else if (mi_rango == 1) {
      MPI_Recv(vector+50, 50, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
   ...
   }
```

**Tipos derivados**

- Si los datos no son de igual tipo, hemos de construir tipos derivados a partir de los primitivos.
- Un tipo MPI derivado es una sucesión de $n$ pares

$$\{(t_0, d_0), (t_1, d_1), \ldots, (t_{n-1}, d_{n-1})\}$$

  donde cada $t_i$ es un tipo básico y cada $d_i$ es un desplazamiento en bytes.
- Un ejemplo podría ser:

```
{(MPI_FLOAT, 0),(MPI_FLOAT, 16),(MPI_INT, 24)}
```

**Tipos derivados**

- La función para construir un tipo nuevo es:

```
MPI_Type_struct(int cuenta,                   /* <== */
                int long_bloque[],            /* <== */
                MPI_Aint desplazamiento[],    /* <== */
                MPI_Datatype lista_de_tipos[], /* <== */
                MPI_Datatype *nuevo_tipo);    /* ==> */
```

**Tipos derivados**

- El parámetro `cuenta` es el número de elementos del tipo derivado y el tamaño de los vectores `long_bloque`, `desplazamiento` y `lista_de_tipos`.
- El vector `lista_de_tipos` contiene el tipo de dato MPI para cada entrada.
- El vector `desplazamiento` contiene el desplazamiento con respecto al comienzo del mensaje de cada entrada.
- El vector `long_bloque` indica cuántos elementos de cada tipo hay en cada entrada.

**Tipos derivados**

- ```
  MPI_Address(void     *variable,   /* <== */
              MPI_Aint *direccion); /* ==> */
  ```

  En principio es equivalente a la sentencia:

  ```
  direccion = &variable;
  ```

  pero nos permite asegurar la portabilidad.
- `MPI_Type_commit(MPI_Datatype *nuevo_tipo); /* ==> */`
  - Mediante este mecanismo, el sistema realiza unos cambios internos en la representación de `nuevo_tipo` para mejorar el rendimiento.
  - Estos cambios no son necesarios si `nuevo_tipo` es usado solo como paso intermedio para construir un tipo más complicado. Por eso esta función va aparte.
  ```

### Tipos derivados simplificados

```
MPI_Type_contiguous(int          cuenta,       /* <== */
                    MPI_Datatype  tipo_viejo,   /* <== */
                    MPI_Datatype *tipo_nuevo);  /* ==> */
```

- El nuevo tipo consiste en cuenta elementos contiguos de un vector de elementos de tipo identificado como `tipo_viejo`.

### Tipos derivados simplificados

```
MPI_Type_vector(int          cuenta,      /* <== */
                int          long_bloque, /* <== */
                int          espaciado,   /* <== */
                MPI_Datatype  tipo_elem,   /* <== */
                MPI_Datatype *tipo_nuevo); /* ==> */
```

- El nuevo tipo consiste en elementos igualmente espaciados de un vector.
- El parámetro `cuenta` es el número de elementos del nuevo tipo, `long_bloque` es el número de entradas en cada elemento, `espaciado` es el número de elementos de tipo `tipo_elem` entre sucesivos elementos del tipo `tipo_nuevo`.

### Tipos derivados simplificados

```
MPI_Type_indexed(int          cuenta,            /* <== */
                 int          long_bloques[],    /* <== */
                 int          desplazamientos[], /* <== */
                 MPI_Datatype  tipo_elem,        /* <== */
                 MPI_Datatype *tipo_nuevo);      /* ==> */
```

- El nuevo tipo consiste en cuenta elementos de tipo `tipo_elem`.
- El $i$-ésimo elemento consta de `long_bloques[i]` entradas y está desplazado en un número `desplazamientos[i]` de elementos de tipo `tipo_elem` respecto del principio.

### Tipos derivados simplificados

- ¿Cómo se realiza la correspondencia entre un tipo de dato enviado y uno recibido?
- Recordemos que un tipo derivado se define como una sucesión de $n$ pares $\{(t_0,d_0), (t_1,d_1), \ldots, (t_{n-1},d_{n-1})\}$ que constituyen la firma del tipo.
- Si la firma del tipo enviado es $\{t_0,t_1,\ldots,t_{n-1}\}$ y la del recibido $\{u_0,u_1,\ldots,u_{m-1}\}$ se debe verificar que

$$\begin{aligned} n &\leq m, \\ t_i &= u_i, \end{aligned}$$

para $i = 0,\ldots,n-1$.

### Empaquetado/desempaquetado

- Podemos empaquetar los datos explícitamente...

```
MPI_Pack(void         *paquete,         /* <== */
         int           cuenta,          /* <== */
         MPI_Datatype  tipo_dato,       /* <== */
         void         *buffer,          /* ==> */
         int           tamanyo_buffer,  /* <== */
         int          *posicion,        /* <=> */
         MPI_Comm      grupo_com);      /* <== */
```

### Empaquetado/desempaquetado

- ... y desempaquetarlos al recibir.

```
MPI_Unpack(void         *buffer,         /* <== */
           int           tamanyo_buffer, /* <== */
           int          *posicion,       /* <=> */
           void         *desempaquetado, /* ==> */
           int           cuenta,         /* <== */
           MPI_Datatype  tipo_dato,      /* <== */
           MPI_Comm      grupo_com);     /* <== */
```

# 4. Topologías e inmersiones

### Topologías e inmersiones

- MPI ve los procesos como dispuestos linealmente y les asigna un identificador correlativo, que llamamos "rango".
- En algunos algoritmos, es más conveniente ver los procesos dispuestos según otras topologías: anillos, cubos, mallas multi-dimensionales, etc.
- MPI dispone de mecanismos para realizar *inmersiones* de sus identificadores en algunos tipos de topología.
- MPI intenta, en la medida de lo posible, realizar la mejor inmersión posible, en función de la topología física de la máquina.

### Topologías virtuales para los procesos

- En general, es conveniente pensar en los procesos como relacionados entre sí mediante un grafo. Cada proceso es un vértice.
- Dos procesos (o vértices) están conectados por un arco si intercambian mensajes.
- La topología más usada es la *malla multi-dimensional*, también llamada *topología cartesiana*.
- Veremos funciones de que dispone MPI para manejar esta inmersión.

### Comunicador cartesiano

- En una topología cartesiana, definimos un nuevo comunicador, de tipo multi-dimensional.
- Cada proceso puede identificarse tanto por su *rango* como por sus *coordenadas*.
- Las coordenadas de un proceso pueden consistir en muchas componentes, tantas como la dimensión de la malla.
- Así podemos manejar procesos de manera "multi-matricial".

**Creación de un comunicador cartesiano**

Permite crear un comunicador con un número de dimensiones igual a `ndimens`, con tamaños para cada dimensión especificados en el vector `dimens`. El vector `periodicos` nos permite especificar si las dimensiones se consideran periódicas (formando un toro) o no; y el parámetro `reordenar` permite o impide que el nuevo comunicador obtenga nuevos valores para los rangos de los procesos.

```
MPI_Cart_create(MPI_Comm  viejo_comunic,   /* <== */
                int       ndimens,         /* <== */
                int       dimens[],        /* <== */
                int       periodicos[],    /* <== */
                int       reordenar,       /* <== */
                MPI_Comm *comunic_cartes); /* ==> */
```

**Relación rango–coordenadas**

- Cada proceso tiene su rango, que es un parámetro necesario en las comunicaciones punto a punto o colectivas.
- Pero en una topología virtual cartesiana cada proceso recibe también unas coordenadas
- Necesitamos unas funciones que nos pasen de uno a otras y viceversa.

**Obtención del rango**

Con esta función obtenemos el rango a partir de las coordenadas:

```
MPI_Cart_rank(MPI_Comm comunic_cartes, /* <== */
              int      coords[],       /* <== */
              int     *rango);         /* ==> */
```

**Obtención de las coordenadas cartesianas**

Con esta función obtenemos las coordenadas a partir del rango:

```
MPI_Cart_coords(MPI_Comm comunic_cartes, /* <== */
                int      rango,          /* <== */
                int      maxdims,        /* <== */
                int      coords[]);      /* ==> */
```

**Obtención de los rangos de procesos específicos**

- Frecuentemente usamos la topología cartesiana para desplazar datos a lo largo de una de las dimensiones de la malla.
- Tenemos una función que, basándose en la dimensión sobre la que deseamos desplazarnos y en un número de saltos, nos calcula los rangos de origen (de donde vamos a recibir datos) y destino (adonde los vamos a enviar).
- Con ellos podemos llamar a la funciones de envío/recepción; típicamente usamos la función `MPI_Sendrecv_replace`.

**Obtención de rangos a lo largo de una dimensión**

Para ello, disponemos de la siguiente función:

```
MPI_Cart_shift(MPI_Comm comunic_cartes,  /* <== */
               int      dir,             /* <== */
               int      paso,            /* <== */
               int     *rango_emisor,    /* ==> */
               int     *rango_receptor); /* ==> */
```

**Obtención de rangos a lo largo de una dimensión**

- El argumento `dir` indexa la dimensión sobre la que queremos desplazarnos. El argumento `paso` nos indica la cantidad a desplazar.
- Los rangos calculados van a parar a `rango_emisor` y `rango_receptor`.
- Si la dimensión es periódica, el cálculo del rango se hace modularmente. Si no, el rango calculado podría caer fuera de la topología, con lo que el valor devuelto sería el valor constante `MPI_PROC_NULL`.

**Partición de comunicadores**

- Con frecuencia necesitamos restringir las comunicaciones (colectivas) a un subconjunto de los procesos del comunicador.

*Ejemplo* 2. Difundir un conjunto de valores a todos los procesos en una columna.

**Partición por «colores»**

El método general de partición emplea la siguiente función:

```
MPI_Comm_split(MPI_Comm comunic,          /* <== */
               int      color,           /* <== */
               int      clave,           /* <== */
               MPI_Comm *nuevo_comunic); /* ==> */
```

Se trata de una función colectiva, que ha de ser invocada por todos los procesos en el comunicador original.

**Partición por «colores»**

- La función `MPI_Comm_split` divide el conjunto de procesos del comunicador original en subconjuntos disjuntos.
- Cada subconjunto contiene justo los procesos que han suministrado el mismo valor en el parámetro `color`.
- Dentro de cada subconjunto, los procesos reciben un rango de acuerdo con el valor del parámetro `clave`.

**Partición de una topología cartesiana**

- Si el comunicador original responde a una topología de tipo cartesiano, es posible realizar particiones a lo largo de una o más dimensiones de la malla.
- Para ello utilizamos la función `MPI_Cart_sub`.

**Partición de una topología cartesiana**

```
MPI_Cart_sub(MPI_Comm  comunic_cart,      /* <== */
             int       nuevas_dims[],     /* <== */
             MPI_Comm *nuevo_subcomunic); /* ==> */
```

El vector `nuevas_dims` se usa para indicar cómo se particiona la topología: si `nuevas_dims[i]` es distinto de cero, el nuevo comunicador retiene la `i`-ésima coordenada (empezando a contar desde 0).

**Ejemplo de sub-particiones cartesianas**

- Consideremos una malla tri-dimensional de tamaño $2 \times 5 \times 7$.
- Si el vector `nuevas_dims` toma como valores $\{0, 1, 1\}$, la nueva topología está hecha de 2 sub-topologías de tamaño $5 \times 7$.
- Si el vector `nuevas_dims` toma como valores $\{1, 0, 1\}$, la nueva topología está hecha de 5 sub-topologías de tamaño $2 \times 7$.

# 5. Operaciones de entrada/salida

**Ideas generales**

- Muchos procesos se están ejecutando simultáneamente, así que es necesario decidir quién efectúa la escritura en la salida estándar y quién la lectura de la entrada estándar.
- La solución más típica es que sólo un proceso se ocupe de ello y
  - reciba de los demás lo que debe ser escrito
  - y envíe a los demás lo que ha leído.

**Soporte de sistema**

- Las funciones normales de C para entrada/salida (`printf`, `scanf`) no están pensadas para el paralelismo.
- No existe un consenso claro acerca de cómo debe funcionar la entrada/salida en sistemas multi-proceso.
- MPI no normaliza nada acerca de la entrada/salida.
- No obstante las implementaciones de MPI suelen proporcionar algún tipo de soporte.

**Proceso de E/S**

- En el caso más sencillo, convertimos las funciones de E/S en operaciones colectivas, utilizando un proceso designado como 'proceso de E/S'.
- Normalmente, todas las implementaciones de MPI:
  - Permiten que al menos un proceso pueda acceder a la entrada y salida estándar y a la salida de error.
  - Para salida, el proceso de E/S recogerá los datos de los demás y los imprimirá.
  - Para entrada, el proceso de E/S leerá los datos necesarios y los difundirá a los demás procesos.
- Así pues, cada comunicador contendrá un proceso designado para la E/S.